DOCUMENT RESUME

ED 330 303 IR 014 917

AUTHOR Kiel, Don; And Others
TITLE NeXT Application Development Workshop. Use and Design of Instructional Applications on the NeXT Computer.
INSTITUTION California State Univ., Los Angeles.
PUB DATE May 90
NOTE 102p.; Funded by a Lottery Dissemination Grant. Notes on a workshop held in May 1990.
PUB TYPE Collected Works - Conference Proceedings (021) -- Computer Programs (101) -- Reports - Descriptive (141)

EDRS PRICE MF01/PC05 Plus Postage.
DESCRIPTORS Authoring Aids (Programing); Computer Graphics; Computer Software Development; Electronic Classrooms; Electronic Mail; Higher Education; Man Machine Systems; *Microcomputers; Multimedia Instruction; *Programing; Programing Languages; *Teacher Workshops
IDENTIFIERS *NeXT Computer

ABSTRACT

Instructional applications for NeXT computers were developed by nine faculty members from the biology, mathematics and computer science, fine arts, chemistry, physics and astronomy, and geology departments as part of a grant awarded to the California State University at Los Angeles. These notes provide a schedule of events and reports from a 2-day workshop at which various features of the NeXTStep environment and their applications were described and demonstrated: (1) "NeXTStep, Mail" (Bob Desharnais, biology); (2) "Slider and Text Field" (Don Kiel, math and computer sciences); (3) "Objective C" (Don Kiel); (4) "Custom View and Slider" (Don Kiel); (5) "PostScript" (Bob Desharnais); (6) "Customizing Your Application" (Tony Longson, fine arts); (7) "Grapher Object" (Don Kiel); (8) "Icon" (Roland Carpenter, physics and astronomy); (9) "MouseDown Events and Bezier Curve" (Don Kiel); (10) "An Animated Dial Object" (Tony Longson); and (11) "DragIt" (Roland Carpenter). Other faculty members involved in the project were Gary Novak, geology, and Trina Valencich, Vickie and Joe Bragin, and Richard Keys, chemistry. A series of 14 screen dumps from some of the applications is attached. (DB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☐ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

ED330303

NeXT Application Development Workshop

Cal State, Los Angeles
Funded by a Lottery Dissemination Grant

BEST COPY AVAILABLE

IR014917

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

Gary Novak

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

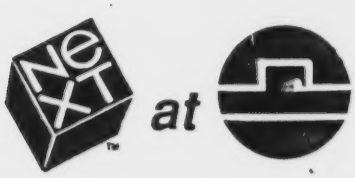
Foreword

Eight NeXT computers were awarded to Cal State Los Angeles in January 1989 as a result of a grant proposal written by Dean Bragin of Natural and Social Sciences. Dean Bragin loaned the machines to individual faculty members from different disciplines who have met bi-weekly since then to share their knowledge about the use of that computer in Instructional Applications. The Faculty members, clockwise from the left, are Bob Desharnais, Biology, Don Kiel, Math and Computer Science, Dean Bragin, Tony Longson, Fine Arts, Dick Keys, Chemistry, Vicky Bragin, Roland Carpenter, Physics, Gary Novak, Chair of Geology; Trina Valencich, Chemistry, is not pictured here.

BEST COPY AVAILABLE

Schedule of Events for Thursday

9.00	*Introductions*	*Gary Novak*
9.10	*An Application*	*Roland Carpenter*
9.30	*NeXT Step , Mail*	*Bob Deshairnais*
10.30	*Break*	
10.45	*Slider and Text Field*	*Don Kiel*
12.00	*Lunch at Choices*	
1.15	*Objective C*	*Don Kiel*
1.45	*Custom View and Slider*	*Don Kiel*
3.00	*Break*	
3.15	*PostScript*	*Bob Deshairnais*
4.30	*Customizing your Application*	*Tony Longson*
5.00	*Dinner*	
7.00	*Demos*	



Schedule of Events for Friday

9.00	*Grapher Object*	*Don Kiel*
10.00	*Icon*	*Roland Carpenter*
10.45	*Break*	
11.00	*MouseDown Events and Bezier Curve*	*Don Kiel*
12.00	*Lunch at Choices*	
1.00	*An Animated Dial Object*	*Tony Longson*
2.30	*Break*	
2.45	*DragIt*	*Roland Carpenter*
3.45	*Closing Remarks*	*Gary Novak*

Multimedia Mail

One unique feature of the NextStep environment is its electronic mail system. Mail messages can contain formatted text, pictures, voice messages, files, folders, and executable applications. Two NeXT workstations on the same mail network can exchange a variety of resources. At the NeXT corporation every professional and clerical employee has a NeXT workstation on their desk. Independent consultants evaluted the NeXT Inc. mail system and claim that the time spent in meetings has been reduced by 70% without any loss in efficiency or productivity.

Multimedia mail could be a very powerful tool in an electronic classroom setting. An instructor could prepare and distribute notes, diagrams, documents, etc. using mail. Voice messages could be used to present the student with proper pronunciations of difficult words. If applications were developed to communicate certain concepts, these could also be included in the mail message. Once a student received a message, they could use the on-line versions of *Webster's Dictionary*, *The Oxford Book of Quotations*, or *Roger's Thesaurus* as reference tools. Students could return assignments by mail. For example, a term paper for a music course could contain portions of the music being analyzed. Electronic mail opens new possibilities for education.

As an illustration, we will construct a simple multimedia mail message. The topic is the human heart. In preparing this mail message, we will use some of the other resources of the NeXT. Step-by-step instructions follow. These instructions assume that certain files already exist and the user knows where to find them.

1. In preparation, place the applications *Mail, Webster,* and *Icon* in the application dock.

2. Launch the *Mail* application by double clicking its icon.

3. Click the send button. Address the message to yourself (your login name). The subject will be "The Human Heart." If your machine is on a network, you might try to send a copy to someone else.

4. Click the mouse in the message area. Type something like "Today's Lesson: The Human Heart" on the first line. Add two carriage returns.

5. Highlight this first line (triple click) and select Font/Font Panel... from the application menu. From the font panel choose Bold Italic font from the Times family and set the font size to 16. Click the Set button to change the font in the mail message. This demonstrates how easy it is to format messages beyond plain ASCII text. Click at the bottom of the message area

6. Now we will create and insert a graphics image. Click the mouse at the bottom of the message area to unhighlight the text and move the insertion point to the third line.

7. Now launch the *Webster* application. Type the word "heart" in the search field and click the define button. A panel with a labeled diagram of the heart should appear on the screen.

8. Launch the *Icon* application from the icon dock.

9. Select Window/New from the *Icon* application menu. You should get a drawing window with the title "Untitled.tiff."

10. Move the panel with the diagram of the heart to a screen location where the heart will not be covered by any of *Icon*'s windows or panels. Click the title bar of the "Untitled.tiff" window to make *Icon* the active application.

11. Select Special/ScreenGrab... from *Icon*'s application menu. Your cursor will change to an upper left corner bracket. Move the cursor to the upper left corner of the heart diagram and drag out a rectangle which encloses the heart but doesn't include any of the text below it. When you release the mouse the heart will appear in the drawing window. This entire step can be repeated if you did not get a good picture the first time.

12. Select Window/SaveAs from *Icon*'s application menu. A Save Panel will appear. Click the "Save Document" button to toggle it to "Save Selection." Click the "Alpha" button to toggle it to "No Alpha." Use the Save Panel's directory browser to choose

an appropriate directory for the tiff file. Change the name of the file from "Untitled.tiff" to "heart.tiff" by typing a new name. Click "OK" or press return to create the new tiff file.

13. Quit the *Icon* application (answer yes, you really want to quit) and quit *Webster.*

14. Click on an exposed portion of the Directory Browser or double click the NeXT icon in the icon dock. Find the new file heart.tiff. (You may need to Window/Update the directory browser.) Select this file so its icon shows at the right of the browser.

15. Drag the icon for heart.tiff into the mail message area. Note that the file automatically images at the insertion point.

16. Below the image of the heart type "1 - aorta ". Leave a couple of spaces after this text.

17. Now we incoporate a voice message. Click the "Lip Service" button. A panel of recorder control buttons will appear.

18. Click record and pronounce the word "aorta". Click stop when you are finished.

19. Click the Play button to hear what your message sounds like.

20. Click the Edit button to see a sonogram of your message. At this point you could use the mouse to highlight a portion of your message for cut, copy, or paste. Normally, you might edit out quiet stretches at the start or end your message.

21. After editing your voice message, click the Insert button to include it with your mail message. A pair of lips should appear.

22. On a new line type "The human heartbeat: " and leave a couple of blank spaces.

23. Return to the directory browser and locate the file "heartbeat.snd". Select this file and drag its icon into the mail message. The sound icon should appear in your message. (In the same way, word processing documents or even entire directories can be included in the mail message.)

24. On a new line type "An EKG simulation: " and leave a couple of blank spaces.

25. Return to the directory browser and locate the file "EKG". Select this file and drag its icon into the mail message. The ekg icon should appear in your message.

26. Click the "Deliver" button to mail the message.

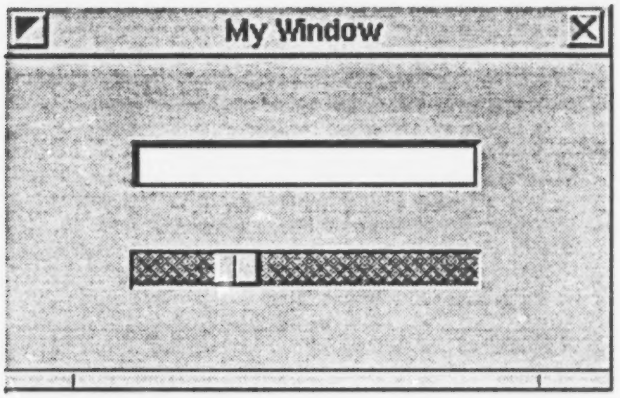
27. After a couple of minutes the mail message should be delivered. You will be informed by a temporary change in your mouse pointer and the letters in the mail icon should flag.

28. Return to the Active.mbox window and click on your message to read it. Double click the lips, sound, and ekg icons and see what happens.

A Simple Example Using the Interface Builder

1. (**Begin application**) Launch the Interface builder, then click the ***Flie*** and ***New Application*** menu buttons.

2. (**Make interface**) Make MyWindow smaller by dragging the bars at the bottom. Drag a slider and a text field to MyWindow from the Palettes window. Delete the word Text in the text field by double clicking on it until it is gray and then pressing the delete key. Make both the slider and the text field longer by dragging one of the knobs at the end. Your window should now look like



3. (**Make connection**) Connect the slider to the text field by dragging the mouse from the slider to the textfield while holding down the Control key. A dark line will follow the cursor and remain when you release the mouse. A Slider Inspector Panel will appear with a column of actions. Highlight the action ***takeFloatValueFrom*** and click the Connect button at the bottom of that panel.

4. (**Test**) To test what we have done so far, click on ***Test Interface*** , the last item in the File submenu. Move the slider back and forth and observe the text field. Exit the test mode by pressing ***Quit*** in your application menu.

5. (**Make connection**) We can also connect the text field to the slider using the same action. Make this connection and test it.

6. **(Disconnect)** This makes an impressive demonstration as it is, but we will now proceed as we normally would in writing our own application. Since we will normally want to define our own actions(methods) we will undo some of what we have done. Quit the test mode and, with the Control key down, click on the text field. The Project director panel will now show the action ***takeFloatValueFrom*** highlighted. Click the ***Disconnect*** button at the bottom of this panel. Do the same for the slider.

7. **(Save)** Returning now to our normal sequence, press the ***Save*** menu button. The Save Panel will ask you for the name of the file. Respond with test /converter (or some other directory and file).

8. **(Create Project)** Click ***File*** and ***Project*** , then click OK in Inspector Window (this creates IB.proj, makefile and the main project file.)

9. **(Form Subclass)** Create a custom sub-class of Object by double clicking the Classes suitcase in the Interface Builder's File window. This will open the Classes window. Click on the left arrow of the browser until you reach the top of the hierarchy, the Object class. Click on the word ***Object*** in the browser. The icon well in the right of the window shows an icon labelled with a ".h" and entitled "Object". Above the icon well is a pull-down menu labelled "Operations". Pull down the menu and select "Subclass". A new icon appears in the icon well labelled "Subclass1". Select the name under the icon, change it to "Converter" and hit return. You have just created a new custom class.

10. **(Add 2 outlets)** Double-click on the name ***Converter*** in your Classes Browser(not the name under the icon) and a Class Inspector will appear for the Converter class. Above the text field near the bottom of the panel there is a two-state button labelled ***Outlets/Actions***. Click it so it is in its ***Outlets*** mode. An outlet is a way to store the address of an object you want to send a message to. Add an outlet called ***mySlider*** (type the name "mySlider" in the text field and hit return) so that you can send messages to the slider. Add another outlet in the same way called ***myTextField*** so that you can send messages to the text field.

11. **(Add 2 actions)** Now let's add the action messages that we want the controls (our text field and our slider) to send when they are clicked by the user. Change the Class Inspector to the ***Action*** mode and enter ***mySliderChanged*** as one of the actions a Converter understands, and ***myTextFieldChanged*** as the second action. Note that Interface Builder adds a colon (":") to your action method name if you don't

type one yourself. All action methods take one argument (called "sender" for the address of the object sending the message) and the colon signifies that.

12. (**Instantiate the new class**) We have described how the class Converter is different from its parent class (or "Superclass") Object. Now let's make an instance of the new class by pulling down the menu labelled ***Operations*** above the icon well in the Classes window and selecting ***Instantiate***. A new icon appears in Interface Builder's File window. It has a sphere on its icon and will be called "ConverterInstance".

13. (**Connect object to outlets**) Make connections between the instance of your Converter object, ConverterInstance and the two interface Objects. Control drag from the ConverterInstance icon in the Files window to the control you want to connect. Select the correct outlet name (***mySlider*** when connecting the slider, ***myTextField*** when connecting the text field) and confirm your selection on the panel by clicking ***Connect*** each time you make a connection.

14. (**Connect controls to object**) Now connect the actions to the object. Control drag a connection from the slider to the icon that stands for your ConvererInstance in the Files window. Select the proper Converter class actions you want the slider to send when it is moved by the user. You should choose ***mySliderChanged:*** for the slider and ***myTextFiledChanged:*** for the text field. Confirm your selection by clicking "Connect" each time you make a connection. Remember that you connect *from* a control to its target.

15. (**Unparse**) Now we need to have Interface Builder write some code. This code will serve as a framework. We will fill in this code to make our converter do its work. To have Interface Builder write a code framework for your application, choose ***Unparse*** from the Operations pull-down menu in the Classes window. This will create two new files called "Converter.h" and "Converter.m". You will then be asked if you want to add these files to the current project. Yes, you do.

16. (**Save**) Save the interface file with the ***File*** and ***Save*** menu buttons.

17. (**Get .h and .m files**) To look at the empty code framework that Interface Builder has written you need to open another program, an application for editing text called

Edit. To do this you can use a short-cut. On the Project Inspector panel, under the "Types" listing, select ".[hm] (class)". The filename "Converter.[hm]" should appear. Double click this filename. This will cause the application Edit to open displaying these two files. Remember, the "Converter.h" file describes the Converter class to the public and the "Converter.m" file contains the code to actually do whatever instances of the Converter class are supposed to do.

18. **(Edit the files)** Change Converter.m by adding the bold-faced lines below:

```
/* Generated by Interface Builder */

#import "Converter.h"
#import <appkit/appkit.h>

@implementation Converter

- setMySlider:anObject
{
    mySlider = anObject;
    return self;
}

- setMyTextField:anObject
{
    myTextField = anObject;
    return self;
}

- mySliderChanged:sender
{
    [myTextField takeFloatValueFrom:sender];

    return self;
}
```

```
- myTextFieldChanged:sender
{
    [mySlider takeFloatValueFrom:sender];

    return self;
}
@end
```

19. (**Save**) Save the changes by clicking on the ***Save*** button in your ***Edit*** menu.

20. (**Compile**)We have defined a custom class, Converter, made an instance of this class and connected some interface objects to it, created empty code framework files to support the new class we defined and filled in two methods in those code frameworks (which are now no longer empty of function). Return to Interface Builder by double clicking its icon. Compile your new program by choosing ***Make*** from the the ***Files*** menu. Test it. Choose ***Make*** from the ***Files*** menu to compile the new application. This will create a new file in the directory you created for this project called whatever you called your Interface Builder file with ".debug" on the end. For example, if you called this interface file "converter", then the executable version of your program would be called "converter.debug".

21. (**Execute**) To test the compiled version, launch it from the Workspace Manager. To do that, select the directory you created for the project. If your new file is not present, update the contents of the Browser's window by choosing ***Update*** from the Workspace Manager's ***Window*** menu. You should now see your new application (with ".debug" tagged to the end of the name you gave your Interface Builder file.) Double click this name in the Browser to launch. Test again.

22. (**Change slider range**) To change the slider so its values range from 0 to 100, return to the Interface Builder and double-click on the slider. Drag the pop-up menu on the Project Directory to the Attributes button and change the maximum value to 100. Save, compile and execute.

23. (**For UNIX users**) Since the shell has been launched, we may also use the shell directly to make and execute the application. Use the ***cd*** command to get to the

directory containing your project, compile, and execute as follows:

```
localhost> cd Workshop/test
localhost> make
make converter "OFILE_DIR = obj"  "CFLAGS =  -O -g -Wimplicit"
mkdirs obj
cc -O -g -Wimplicit  -c Converter.m -o obj/Converter.o
cc -O -g -Wimplicit  -c converter_main.m -o obj/converter_main.o
cc -O -g -Wimplicit -segcreate __ICON __header converter.iconheader -
segcreate __ICON app /usr/lib/nib/default_app_icon.tiff -segcreate
__NIB converter.nib converter.nib -o converter obj/Converter.o
obj/converter_main.o -lNeXT_s -lsys_s
localhost> converter
```

Object-oriented Programming

In object-oriented programming, an object is a specific instance of a class of objects. The class abstractly describes the common characteristics of all objects in the class. A non-programming example is the class of dolphins with Flipper as a particular object in that class.

Object-oriented programming uses these principles:

• encapsulation: an object is made up of a data structure and the operations performed on that data structure

• inheritance: classes of objects inherit all the data structures and operations of their ancestors

• messaging: objects interact through messages which ask them to perform one of their defined operations. Classes may also receive messages.

Encapsulation

Two analogies:

A capsule holds together different ingredients of a medication. OOP encapsulates the data structure (made up of instance variables) and the operations (called methods) which operate on those variables. This allows data hiding or data protection.

An IC (integrated circuit) encapsulates a number of functions and allows access to the circuitry only through pins. The only way a programmer can operate on the instance variables is by using the defined methods .

To implement this, Objective-C applications have two files:

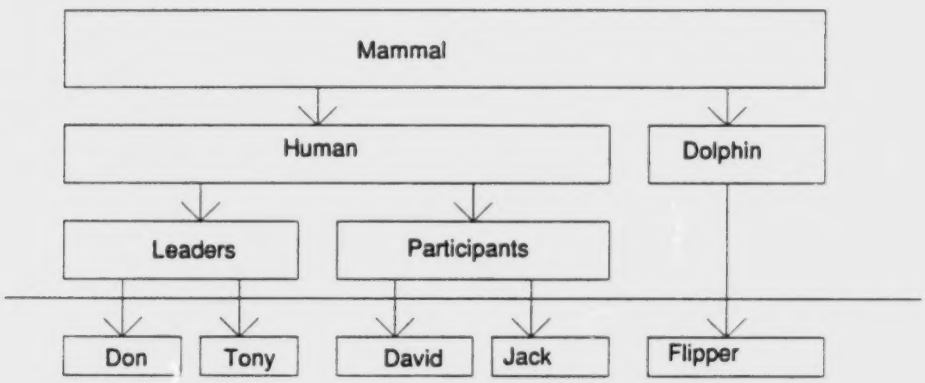
• the .h file is called the interface file and lists only the names and types of the instance variables and the names of the methods with type declarations of the arguments used by the methods. This is sometimes called the public file.

• the .m file is called the implementation file and contains the code which implements the methods. This is sometimes called the private file.

Inheritance

One important feature of OOP is that classes inherit all instance variables and methods from their ancestors.

A simple illustration



Each of the classes (above the horizontal line) inherits all the characteristics of its superclass. Each of the instances of classes (below the line) is a specific example (instantiation) of its class. Communication, or messaging, between the instances in the example above may be informal and may be done many ways - speech, hand signals, telephone, electronic mail, fax, etc. In object-oriented programming the messaging protocol is more disciplined .

Messaging

In OOP, instead of calling functions with parameters, we send messages to objects. These messages may have arguments.

In Pascal, we might use statements like

```
moveParallelogram(x, y);
moveSquare(x, y);
area := squareArea(mySquare);
area := parallelogramArea(myParallelogram);
```

In Objective-C we will use message statements of the form

```
[mySquare move :x :y];
[myParallelogram move :x :y];
area = [mySquare area];
area = [myParallelogram area];
```

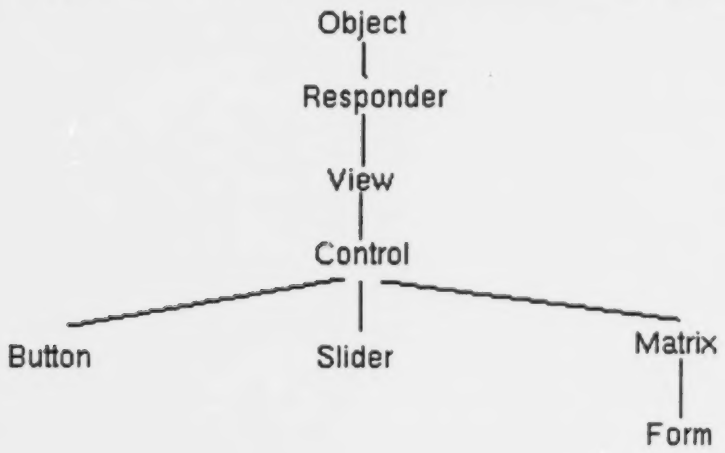
The first word in the message statement is the name of the object to which the message is sent. The second word is the name of the message (method) and arguments are preceeded by colons.

The ability to use the same name for methods which will be applied to different classes is called polymorphism. This property, used with inheritance, simplifies the creation of reusable code.

Specific examples from NextStep

NextStep has defined a heirarchy in its AppKit. The most basic class in the Appkit is called the Object class. One of its subclasses is the Responder which in turn has a subclass View. View has a subclass Control which has the subclasses Button and Slider as well as a subclass Matrix, which has a subclass Form.

The appropriate part of the heirarchy is contained in the diagram below



The entire Appkit heirarchy is found on page 46 of
/NextLibrary/Documentation/NeXT/SysRefMan/22_ClassSpecs/AppKitIntro.wn

Messages to Button

Messages in object-oriented programming are analogous to procedure calls in procedural languages in that they invoke the operation of a certain block of code. These messages may also contain arguments which are analogous to parameters in procedural languages.

Assume that we have used the Interface Builder to create a button named myButton, we may send the following message to that button :

```
[myButton setTitle:"start"];
```

This method is used to change the title written on the button.(The default is "Button").

Another example is

```
[myButton setIcon:picture];
```

Look up Button in

/NextLibrary/Documentation/NeXT/SysRefMan/22_ClassSpecs/AppKit/Button

to see what other kinds of messages one may select.

Forms in NextStep

Again, suppose we have used the Interface Builder to create an object named myForm of the class Form.

If the form we are dealing with is one of a vertical Matrix of Forms, we use

```
[myForm setIntValue:1.0 at:0];
```

to set the value of the 0th(first) form in the matrix to 1.0.

Find this method in the manual.

If we have only one Form, we would use

```
[myForm setFloat Value:1.0];
```

or

```
[myForm intValue];
```

This is an example of how the heirarchy is used in the AppKit, since these methods are not defined in Format all but in Control !!

Find these methods in the manual.

Instantiation

To make use of these abstract classes we need to create an object, or a specific member of the class. A specific object is called an instance of its class and the process of creating an object is called instantiation. To avoid confusion at this point, we will introduce notation commonly used in object-oriented programming. Class names, like Button, will be capitalized. The names of objects will not be capitalized and a specific Button (object) might be instantiated in Objective-C with a statement like

```
myButton = [Button new];
```

Note: AppKit objects will usually be instantiated by the Interface Builder.

In Objective-C implementation, myButton will be the id (identifier) of a specific member of the Button class. The right hand of this statement, `[Button new]`, is called a "message expression." The first word, Button, in the message expression is the name of the class or object to which the the message is sent. while the second word new, is the message to be sent (technically the selector of the method to be used.)

Factory Methods and Object Methods

You may have noticed that some methods begin with + and some with -.

Those which begin with - are called object methods since they are used with specific objects which have been instantiated.

Those beginning with + are called class methods or factory methods since they are messages to classes which will actually instantiate objects of that class.

For example,in the Button class,there are methods like
`+new` and `+newFrame:` as well as others.

We do not have a specific instance of a button until we send a message like

```
myButton = [Button new];
```

for the default case, or

```
myLargerButton = [Button newFrame:r];
```

where r is a description of the rectangle in which we wish to draw the button.

In the case of Appkit classes, we may also use the Interface Builder to create objects of different sizes.

PostScript

PostScript is a page description language used to provide a device-independent standard for representing the printed page.

An intuitive way of thinking of Postscript is to imagine a pen tracing a path over a page.

The most useful instructions are

`PSmoveto(x,y)` moves the pen to the point (x,y) without tracing a path

`PSlineto(r,s)` traces a path from the present pen position to (r,s)

`PSsetgray()` sets the darkness of the ink ranging from white(0) to black(1)

`PSsetlinewidth()` sets the width of the line default width is 0 (one pixel)

`PSstroke()` actually draws the path the pen has traced out

`PSfill()` fills in the path traced out

For example, to draw a rectangle, we might use

```
PSsetgray(0.0);                 //0 is black
PSsetlinewidth(0.0);            //default is one pixel
PSmoveto(0.0, 0.0);
PSlineto (100.0, 0.0);
PSlineto(100.0, 50.0);
PSlineto(0.0, 50.0);
PSlineto(0.0, 0.0);
PSstroke();
```

Another way of drawing a rectangle is

```
PSsetgray(NX_BLACK);
PSsetlinewidth(0.0);
PSrectstroke(0.0, 0.0, 100.0, 50.0);
```

For more information on PostScript, see the three Addison Wesley books:

PostScript Language Tutorial and Cookbook

PostScript Language and Reference manual

PostScript Language Program Design

Drawing

We will now study some different methods of drawing. In our first example, we will draw directly to the customView.

1. (**Create application and project and save**) Launch the Interface Builder, create a ***New Application*** and ***save*** it in test2/testDraw. Press the ***Project*** menu button to create a Project file.

2. (**Create interface**) Drag a slider and a customView into the window.

3. **(Subclass)** Open the Classes Browser by double-clicking on the Classes suitcase in the File Window. Since we will define the customView as a subclass of a View, highlight ***View*** in the Classes Browser. (View is a subclass of Responder which is a subclass of Object.) Click on the ***Operations*** menu item in this window and drag down to the Subclass button in the menu and release the mouse. Rename subclass1 as DrawView

4. (**Add action**) Add getSlider as an action for the Draw View subclass.

5. **(Instantiate as a View)** We do not instantiate this new subclass the same way as before. In this case we first double-click on the customView and drag the pull-down menu of the Project Inspector up to ***Attributes***. Note that DragView is one of the View classes listed. Highlight DragView and press the OK button at the bottom of the Inspector Panel.

6. (**Make connection**) Connect the slider directly to the CustomView using the action getSlider.

7. (**Unparse**) ***Unparse*** as before.

8. (**Save**) ***Save*** the interface file.

9. (**Modify code**) Copy the following bold-face text from
Workshop/Programs/Draw1/DrawView.m
and paste it to your DrawView.m

```
#import "DrawView.h"
#import <appkit/Slider.h>
#import <dpsclient/wraps.h>

@implementation DrawView

float myFloat; //this should be instance variable in DrawView.h

+newFrame:(const NXRect *) frm
{
     self = [super newFrame:frm];
     return self;
}

-drawSelf:(NXRect *)r :(int)c
{
    PSsetgray(1.0);                    //clear the view
    NXRectFill(r);

    PSsetgray(0.0);                   //draw the circle
    PSarc(bounds.size.width*myFloat,bounds.size.height/2.0,
              50.0,0.0,360.0);
    PSfill();
    return self;
 }

- getSlider:sender
{
    myFloat = [sender floatValue];
    [self display];
    return self;
}
@end
```

10. **(Examine the code)** When this application is launched, newFrame is automatically invoked, instantiating self as a specific object DrawView, a subclass of

View. Then drawSelf is also automatically invoked, drawing the background rectangle and the circle. Each time the slider is moved, the message getSlider is sent to DrawView.The getSlider method send a display message to self, the DrawView, and the display method sends a drawSelf message!

In drawSelf, **r** is a pointer to a structure of the NXRect type. (NXRect structures are defined on page 4-15 of the Systems Reference Manual.) The variable **bounds** is the name of the smallest rectangle which contains the view. It has two subfields- **origin** and **size** and size has the subfields **width** and **height**, which we use in drawSelf.

11. (**Add Printing Capability**) Click on the rightmost icon at the top of the Palettes panel of the Interface Builder. Drag the meu button labelled **Item** to the menu for your application. Double-click on the word **Item** until it is gray and then replace it by the word **Print**. Connect the new menu button to theCustomView using the ***printPSCode:*** action which will automatically appear in the Actions column of the Project Inspector. Save, compile, and test if you have printer.

Using Compositing

By way of contrast we will use the compositing feature available in NextStep to do more sophisticated drawing. We will draw our "background" on the screen and form an off-screen bitmap on which we will draw the moving part of our picture. Then each time the application receives a message from the slider, it will draw a new circle on the off-screen bitmap and composite this to the screen. We will also set the alpha channel to be half transparent (and half opaque) so that we will be to see the background through the circle. Execute **testDraw** in **Programs/Draw2** to see that this is done.

The differences in the code from the previous exmple are in boldface.

```
#import "DrawView.h"
#import <appkit/Slider.h>
#import <dpsclient/wraps.h>
#import <appkit/Bitmap.h>

@implementation DrawView
float myFloat;  //should be instance variable in Drawview.h

id offWindow; //should also be instance variable

+newFrame:(const NXRect *) frm
{
     self = [super newFrame:frm];

     //offWindow is declared as an id in testComp.h and is
     //defined as a bitmap the same size as the CustomView
     offWindow = [Bitmap newSize       :frm->size.width
                                       :frm->size.height
                                       type:NX_UNIQUEBITMAP];
     [offWindow setFlip:NO];

     return self;
```

```
}

-drawSelf:(NXRect *)r :(int)c
{
    PSsetgray(1.0);                    //clear the view
    NXRectFill(r);

    PSsetgray(0.67);                   //draw some "background"
    PSrectfill(10.0, 10.0, bounds.size.width-20.0,
               bounds.size.height-20.0);

    //remove three drawing statements and move to getSlider:
    [offWindow composite:NX_SATOP   toPoint:&r->origin];
    return self;
 }



- getSlider:sender
{
    myFloat = [sender floatValue];
    [offWindow lockFocus];

    //start with a clear (transparent) bitmap
    PScompositerect(0.0, 0.0, bounds.size.width,
                    bounds.size.height,NX_CLEAR);

    PSsetalpha(0.5);                   //set alpha channel
    PSsetgray(0.33333);
    PSarc(myFloat*bounds.size.width,bounds.size.height/2.0,
          50.0,0.0,360.0);
    PSfill();

    [offWindow unlockFocus];

    [self display];
```

```
    return self;
}

@end
```

Comment: Note that the drawing here is done by compositing. Instead of drawing directly to the Custom View, we have created a bitmap called offWindow in the newFrame: method by the statement

```
offWindow = [Bitmap newSize          :frm->size.width
                                     :frm->size.height
                                     type:NX_UNIQUEBITMAP];
```

The dimensions of offWindow are exactly the same as those of the Customview. Then when we do the drawing in getSlider, we first use

```
[offWindow lockFocus];
```

to make sure we draw to offWindow. We then draw a clear rectangle

```
PScompositerect(0.0, 0.0, bounds.size.width,
                bounds.size.height,NX_CLEAR);
```

We draw the circle in the proper location with

```
PSsetgray(0.33333);
PSarc(myFloat*bounds.size.width,bounds.size.height/2.0,
      50.0,0.0,360.0);
PSfill();
```

and then use

```
[offWindow unlockFocus];
[self display];
```

after drawing. The display method invokes the drawSelf: method which first draws some "background" with

```
PSsetgray(0.67);
PSrectfill(10.0, 10.0, bounds.size.width-20.0,
            bounds.size.height-20.0);
```

and then composites the bitmap to the View by the statement

```
[offWindow composite:NX_SATOP  toPoint:&r->origin];
```

Each time the slider is moved, the ball is drawn at the proper place in the bitmap. Then when getSlider sends the display message, drawSelf is invoked and the "background" is drawn and the bitmap offWindow is composited to view. Although we could draw everything directly to the View, compositing is useful when there are different "layers" of images on the screen. We have used the NX_SATOP compositing operation here to composite one image over the other. This is one of 13 different compositing operations described on page 4-41 of the Refernence Manual.

Of course, if the background were more complicated we could also draw that once on another bitmap and composite that to the screen rather than draw it each time. This kind of drawing is especially useful in animation when only a small part of the entire picture is changed for each frame.

Since in this example the circle always remains the same size, a different (and more efficient) approach to drawing the circle is to create a small bitmap which will just contain the circle and composite that to different points on the screen to give the effect of motion. To implement this we could have newFrame invoke a new method which would draw the bitmap containing a circle only once. Then we could compute (or read from the slider) the coordinates of the NXPoint pt which would represent the lower left corner of the circle's bitmap and use the message

```
[offWindow composite:NX_SATOP   toPoint:&pt];
```

to composite the circle in the proper location.

Comment: We have taken advantage of the availabilty of the alpha channel in this application as well. The statement

```
PSsetalpha(0.5);
```

sets the alpha value to 0.5 which means that it is half transparent and half opaque. This is obvious when when the slider is moved so that the circle overlaps the edge of the rectangle forming the background.

Using PostScript in Applications

Graphical images are created using the PostScript language. Within an application, there are three ways of executing PostScript commands. First, PostScript code can be saved in an ASCII file and executed from within a program. This method is analogous to the one used in the application Yap. We've already seen many examples of the second method: the predefined PSfunctions() made available in C. The third method is a hybrid of the first two. We can define our own PSfunctions() in a "psw" file and call them from C. We shall now look at a application implemented using all three techniques.

Random Walk Using PSrun()

The PSrun() function can be used to simply feed PostScript code from an ASCII file to the window server. The following PostScript code can be found in the file randwalk.ps. The lines preceded by % are comments.

```
%  This PostScript code simulates a random walk within a circle.
%  At each step a new direction (angle) and stepsize are chosen.
%  The angle is chosen from a uniform distribution with a range
%  + or - openangle/2; thus the user can "restrict" the new direction
%  by setting the parameter openangle.  The stepsize is chosen from
%  a negative exponential distribution with an average stepsize of
%  avgstep; this value can also be modified by the user.
%  The random walk begins in the center continues until it hits
%  the boundary of the circle.  For values of avgstep < 2 and
%  openangle > 300, this may take a little time.

% User can change these parameters.
/avgstep 2 def      /openangle 360 def   /currentangle 90 def

% Polar (radius angle)  -->  rectangular (x y) coordinates
/polar2rect { dup 3 -1 roll dup 4 1 roll exch cos mul
```

```
            3 1 roll sin mul} def

% Uniform random number between 0 and 1
/urand {rand 2147483647 div} def

% Negative exponentially distributed random number
%  (mean for distribution must be on top of stack)
/nexprand {1 urand sub ln mul neg} def

% Distance from origin (x y must be on top of stack)
/dist { dup mul exch dup mul add sqrt} def

% Takes one random step (changes currentangle and currentpoint)
/takestep {avgstep nexprand urand 0.5 sub openangle mul
               currentangle add dup /currentangle exch def
               polar2rect rlineto currentpoint stroke moveto } def

%  End of Prologue

1 setlinewidth
0 setgray
250 250 translate
0 0 200 0 360 arc clip stroke      % define clip area and draw circle
0 0 moveto

% step until hit circle
{takestep flushgraphics currentpoint dist 200 ge {exit} if} loop
```

Now launch the application PSrun. It consists of a single window with a 500x500 custom view and two new menu items: File/Load and File/Execute. Click the File/Load menu button to obtain an Open Panel browser. Select the file randomwalk.ps. Now click the File/Execute menu button. The random walk simulation is a dynamic execution of the above PostScript program.

To see how this PostScript program was used in the PSrun application, examine the

following code from the implementation file PSrunview.m.

```
/* Generated by Interface Builder */

#import "PSrunview.h"
#import <dpsclient/wraps.h>
#import <appkit/Window.h>

@implementation psrunview

+ newFrame:(NXRect *)rect
{
   self = [super newFrame:rect];        // instantiate view
   selected = 0;                        // initialize flag at start-up
   return self;
}

- drawSelf:(NXRect *)rects :(int)rectCount
{
   NXRect r;                    // rectangle structure for view bounds
   char ps_window_title[120]; // character vector for window title

   [self getBounds:&r];                    // get bounds of view
   PSsetgray(NX_WHITE);                    // set color to white
   NXRectFill(&r);                         // fill view with white
   if (selected) {
      strcpy(ps_window_title,"Executing:  ");     // copy into string
      strcat(ps_window_title,[ps_file filename]); // add file name
      [window setTitle:ps_window_title];  // change window title
      [NXApp updateWindows];              // update window title
      PSrun([ps_file filename]);          // EXECUTE PostScript FILE
      strcpy(ps_window_title,"Current File:  "); // modify string
      strcat(ps_window_title,[ps_file filename]); // add file name
      [window setTitle:ps_window_title];  // change window title
```

```
        [NXApp updateWindows];                  // update the window
    }
    return self;
}

- psrun:sender
{
    [self display];    // Use display method to execute PostScript file
    return self;
}

- getfile:sender
{
    char ps_window_title[120];              // character vector
    const char *const types[2]={"ps",NULL};      // list of file types

    ps_file = [OpenPanel new];              // instantiate Open Panel
    [ps_file runModalForTypes:types];       // get PostScript files
    strcpy(ps_window_title,"Current File:  ");    // copy into string
    strcat(ps_window_title,[ps_file filename]);  // catenate file name
    [window setTitle:ps_window_title];      // change window title
    [NXApp updateWindows];                  // update window title
    selected = 1;                           // file is selected
    return self;
}

@end
```

The getfile: method creates an Open Panel for loading in the PostScript file. The drawself:: method implements the PSrun() command which takes the PostScript file as its argument. The psrun: method sends a display method to the view which automatically causes drawself:: to be called. Much of the code is simply involved with changing and updating the window title.

One advantage of the PSrun() command is that the PostScript code which creates

the graphical output is determined at run time. Use File/Load and File/Execute to see the output of the scribble.ps and fireworks.ps files.

Random Walk Using PSfunctions

A disadvantage of the PSrun() technique is that is does not allow us to easily modify the graphical output. This is because all of the parameters determining the behavior of the graphics commands are isolated in the PostScript file. Fortunately, NeXT's implementation of object-oriented C comes with a library of C functions which parallel the PostScript operators. For example, the PostScript operator "moveto" can be implemented using the C function PSmoveto(x,y). The application RandWalk1 relies entirely on these "PSfunctions" to simulate the random walk. Launch this application to see how it behaves. You can easily modify the behavior of the simulation using sliders to change parameters. (Note: stepsizes less than one with large open angles will lead to simulations that take a long time to complete.)

Examine the code in the implementation file RWview.m to see how this program works:

```
/* Generated by Interface Builder */

#import "RWview.h"

@implementation RWview

+ newFrame:(NXRect *)rect
{
    self = [super newFrame:rect];
    openangle = 180.0;                      // initialize open angle
    avgstepsize = 5.0;                      // initialize avg. step size
    srand((unsigned int)(time(NULL)/2));    // seed random numbers
    return self;
}
```

```
- drawSelf:(NXRect *)rects :(int)rectCount
{
    NXRect r;
    float x, y;
    double angle, stepsize, radius=0.0, max_num=2147483347.0;
    double currentangle=90.0;

/* Initialization:  erase old view, set clip path, draw circle,
    make origin current point */

    [self getBounds:&r];                    // get window rectangle
    PSsetgray(NX_WHITE);                    // set color to white
    NXRectFill(&r);                         // fill view with white
    PSsetgray(NX_BLACK);                    // set color to black
    PSsetlinewidth(1.0);                    // set line width to 1 pixel
    PStranslate(250.0,250.0);               // translate the origin
    PSarc(0.0,0.0,200.0,0.0,360.0);         // path for circle
    PSclip();                               // set clip path
    PSstroke();                             // draw circle
    PSmoveto(0.0,0.0);                      // make origin current point

/* MAIN LOOP BEGINS HERE */

if (runflag && (avgstepsize>0.0)) do {

/* Compute random angle uniformly distributed between + or -
(openangle/2) and step size from a negative exponential distribution
with mean avgstepsize */

      angle = (rand()/max_num - 0.5)* openangle;
      stepsize = - avgstepsize * log(1 - rand()/max_num);

/* Compute next coordinates and update current angle */

      currentangle = currentangle + angle;        // new current angle
```

```
        x = x + stepsize*cos(currentangle/180.0);  // new x coordinate
        y = y + stepsize*sin(currentangle/180.0);  // new y coordinate

/* Take one step and update current point */

        PSlineto(x,y);          // line from current point to new point
        PSstroke();             // stroke new line segment
        PSflushgraphics();      // update display
        PSmoveto(x,y);          // update current point

/* Compute distance from origin */

        radius = sqrt(x*x + y*y);

/* MAIN LOOP ENDS HERE */

    } while(radius<200.0);
    return self;
}

- execute:sender
{
    runflag = YES;          // set flag to execute random walk
    [self display];         // will call drawSelf method
    runflag = NO;           // reset flag
    return self;
}

- newangle:sender
{
    openangle = [sender doubleValue];          // get new open angle
    [anglefield setDoubleValue:openangle];     // display new open angle
    return self;
}

- newstepsize:sender
```

```
{
    avgstepsize = [sender doubleValue];          // get new step size
    [stepfield setDoubleValue:avgstepsize];      // display new step size
    return self;
}

- setAnglefield:anObject
{
    anglefield = anObject;
    return self;
}

- setStepfield:anObject
{
    stepfield = anObject;
    return self;
}

@end
```

Again, all the graphics code can be found in the drawself:: method. The execute: method calls the display: method of the view which in turn calls the drawself:: method. Much of the computational work which was done in PostScript using PSrun() can now be done using ordinary C statements. Most importantly, the instance variables avgstepsize and openangle can be changed using sliders before each simulation.

Random Walk Using PSwrap functions

There is yet a third way to combine PostScript and C. It is possible to define your own "PSfunctions" and associate these functions with your application using an ASCII file with a .psw file extension. One advantage of this technique is that it makes for much clearer C code. Long sequences of PSfunctions are avoided. Another

advantage is that this technique makes for faster execution. The compiler turns the pswrap functions into binary coded PostScript which the window server can deal with more quickly. Finally, these pswrap functions can be reused in other applications.

The application RandWalk2 is very similar to RandWalk1 except that it uses two pswrap functions. These are defined in the file RWpswraps.psw. Note that this file must be included in the project using the Interface Builder. Here is what this .psw file looks like.

```
/* PSwrap functions for RandomWalk2 */

/* Initialization function:  erase old view, set clip path, draw
circle, make origin current point */

defineps  Initialize()
   1 setgray
   0 0 500 500 rectfill
   0 setgray
   1 setlinewidth
   250 250 translate
   0 0 200 0 360 arc
   clip
   stroke
   0 0 moveto
endps


/* Take one step and update current point */

defineps  TakeStep(float x,y)
   x y lineto
   stroke
   flushgraphics
   x y moveto
endps
```

Note that these functions are defined using standard PostScript code. Arguments can be passed in the function call. These arguments act as standard key values in PostScript. Although there is no need for it here, pointers can be used to return altered argument values to the calling C program.

Only a few changes need to be made to the implementation file for RandWalk2. First the following line must be included:

```
#import "RNpswraps.h"
```

Although it seems strange to import a file which doesn't exist, this file will be created by the C compiler in its first pass through the program code. The only changes necessary from the RandWalk1 implementation file are in the drawself:: method. This method appears below with the modifications highlighted with italics:

```
- drawSelf:(NXRect *)rects :(int)rectCount
{
   NXRect r;
   float x, y;
   double angle, stepsize, radius=0.0, max_num=2147483347.0;
   double currentangle=90.0;

/* Initialization:  erase old view, set clip path, draw circle,
   make origin current point */

   Initialize();      // use pswrap function to initialize

/* MAIN LOOP BEGINS HERE */

if (runflag && (avgstepsize>0.0)) do {

/* Compute random angle uniformly distributed between + or -
(openangle/2) and step size from a negative exponential distribution
```

```
with mean avgstepsize */

      angle = (rand()/max_num - 0.5)* openangle;
      stepsize = - avgstepsize * log(1 - rand()/max_num);

/* Compute next coordinates and update current angle */

      currentangle = currentangle + angle;        // new current angle
      x = x + stepsize*cos(currentangle/180.0);   // new x coordinate
      y = y + stepsize*sin(currentangle/180.0);   // new y coordinate

/* Take one step and update current point */

      TakeStep(x,y);          // use pswrap function to draw next step

/* Compute distance from origin  */

      radius = sqrt(x*x + y*y);

/* MAIN LOOP ENDS HERE */

   } while(radius<200.0);
   return self;
}
```

As a general rule, use pswrap functions whenever you would have six or more PSfunctions appearing together in your program. Your program will be easier to read and will run a little faster. The increased performance is not noticable in the case of RandWalk2 *versus* RandWalk1 because most of the time is still spent flushing the graphics to the view.

Customizing an Application from Interface Builder

Components of an Application

A typical application consists of a window and a menu. The window may have a view and fields to display or enter information, and buttons or sliders to initiate actions or change values. The menu (like a restaurant menu) allows choices, and provides information about those alternatives.

The look of the application is important. If it is designed well it will be easy to use and a pleasure to use.

Interface Builder helps you design the Application Interface and make changes to it. The tools are easy to use, you can drag an object into your application from the Palettes, and change the size or position of it.

Additional control over the way the Interface looks is provided by the **Inspector**, where you can explicitly change the color, size, position or other attributes of an object, **Layout** where you can make changes to groups of objects, and **Font** where you can change the attributes of the text in your application.

InterfaceBuilder also lets you include images or sounds in the application from elsewhere. In the classes panel of your interface there are suitcases called Icons and Sounds. Double clicking them to open them up shows the defaults that are available for use in your application, but you can add your own sounds or icons by dragging them from the Directory Browser's icon well into the appropriate suitcase. Icons and sounds are usually associated with buttons.

Using Interface Builder's Design tools

Open a new application. Resize the window — it is probably not the right size anyway. How big should it be? How do you want it to look in comparison to other

apps open at the same time?

With the window selected go into the Inspector, change its name in the Title field. Decide if you want miniaturize, close or resize attributes. If you have cancelled the resize option and decide at a later date that you want to change the size of the window, the Layout menu item provides an easy way to do so.

Miscellaneous in the window Inspector's pop up menu lets you explicity set the size and position of the window (useful if you have a bitmap of some fixed size that you want to use as a background to a view, or if you want to make a graph with some specific aspect ratio).

Choose Layout and Align from the menu and select Set Grid On, and Show Grid. Drag a customView from the Palette and drop it into the Window. It snaps to the grid. If you resize it by dragging one of its handles, the new size will also conform to the grid.

Drag a button to the window. Holding down the Alternate key, drag on the lower right handle to make lots of buttons in a row, column or grid. The new buttons will have the same characteristics as the first (size, font, etc.) so you should decide what the first button looks like before you replicate it.

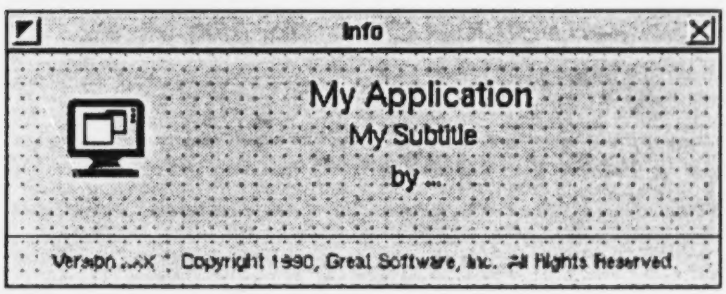
Sometimes it is convenient to group some of the objects in the window. Objects can be grouped temporarily to move them to some other location conveniently, or they may be grouped to show some functional affiliation. Drag some objects into the window (a button and text field, for example). Click on one object to select it, and then, holding the Shift key down click on the other objects that you want to group to select them. With these objects selected go to the Layout menu and choose Group. The objects are surrounded by a box with a title. When the box is selected, the Inspector Panel will let you change some of its attributes. Even though the objects are grouped you can still alter the characteristics of individual objects by double clicking on them to select them, or you can ungroup them.

Drag a button into the window. Click on the Sounds suticase in the classes panel, choose one ot the default sounds and drag it to the button. The sound is associated with the button, so that it will play each time the button is pressed. Choose Test

Interface from the File menu to verify this. If you are not sure which sound you want, you can try them out by double clicking on them. This will open the Sound Inspector and let you play the sound. You can add your own sounds to those available to the application by dragging them from the Directory Browser to the Sounds suitcase in Interface Builder.

To add an icon to the button you can open the Icon suitcase and drag an icon to the button. If you want to add your own icon (that you have made in Icon) find the Tiff file in the Directory Browser and drag the tiff file to the Icon suitcase, which will open to accept it. Now that Interface Builder knows about it, you can drag it to any of the buttons in you application. The Info panel is a good place to use a custom icon.

To display the Info panel, double click on the Info Panel icon in the classes panel. The Info Panel is a good place to tell your users who you are and, briefly, what the application is about.



When you complete your application, it is really nice to give it a custom icon that will identify it in the dock or browser. The icon must be exaclty 48 x 48 pixels square, and must be saved with an Alpha channel (even if it doesn't use it). Open the Project Inspector and click on Attributes.

With the Application icon selected, click the Set button and find the tiff file that contains your icon. From the Shell, go to the directory where your application is, and type `make install` which will compile the application and install a copy of it in your local Apps directory (/me/Apps for example) where it will show up with your custom

icon. If it doesn't show up with the custom icon select Utilities/Find Applications from the Workspace menu. If it still doesn't show up, check that the icon is 48x48 and saved with the Alpha channel. If it still doesn't show up call NeXT!

Look in the Apps directory to see some of the custom icons developed by Don Kiel and Bob Deshairnais using images that were grabbed by Scene from the running application.

Guidelines for Designing an Application's Interface

— aim for clarity and simplicity, avoid cluttering up windows with too many things

— make good use of space, don't feel as though you should fill up the whole window

— think of the size and positon and look of your application's window in relation to other things that may be on the screen

— use the alignment tools to make things look neat

— use a wide range of tonality, use contrasts of dark and light areas

— design your applications with color in mind, PostScript lets you draw in color now, so that your applications will run without modification when color machines are available

— use size and position (and color) to indicate the relative importance of things

— don't mix font families, try to use just two or three point sizes from one font family

— use self evident icons, avoid obscure titles or labels

— use sound to reinforce a user's actions, avoid boring or repetitious sounds

— imitate the look of Applications designed by people at NeXT

— think of ergonomics, are things easy to find? does the user have to move the mouse too much?

— have someone else test your interface, listen to their comments and be prepared to make changes to the design

— make sure that everything works!

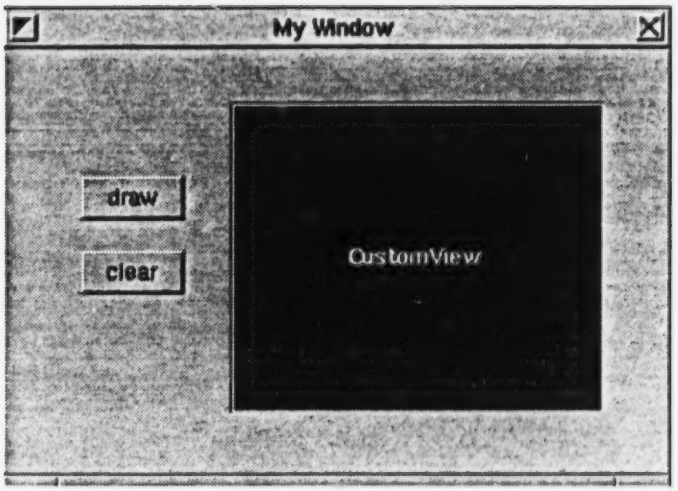
Use of An Existing Class

One of the advantages of object-oriented programming is that objects which have been created and thoroughly tested can be reused by other applications. In this example we will use a Grapher class which we have previously created. The instance variables and the methods of this class are defined in the file Grapher.h. What do you suppose the methods will do when they are invoked?

```
#import <appkit/View.h>

@interface Grapher:View

{
    float minX, minY;
    float viewWidth, viewHeight;
    float ratiox, ratioy;
    id offWindow;
}

-drawAxes:(float)minx :(float)maxx :(float)miny :(float)maxy;
-drawLine:(float)x1 :(float)y1 :(float)x2 :(float)y2;
-drawPoint:(float)x1 :(float)y1;
-clear:sender;

@end
```

Normally you will do steps 1 through 5 below but to save time we have done them for you in **Workshop/Partial/Grapher**. Find **testGraph.nib** in that directory in the Directory Browser and double-click on it. Note that the Interface Builder is launched with testGraph.nib as the active application. Now proceed to step 6.

1. (**Create Interface**) To demonstrate the use of this class, begin by constructing a graphical interface similar to this:



2. **(Save and create project)** Save as usual in a file named testGraph and create a project directory.

3. **(Subclass)** Now we proceed to build our application. In your classes Browser , work your way back up the tree to the Object Class and make sure it is the only class highlighted in the Browser. Then use the pull-down menu to subclass Object and rename the default Subclass1 to TestGraph.

4. **(Add an outlet and an action)** Double-clicking on the name TestGraph will cause the class Inspector for TestGraph to appear. Add the outlet **myGraph** and the action **drawIt** by typing these names in the Inspector window with the switch set to "outlet" and "action" respectively.

5. **(Unparse and Instantiate)** Now return to the Classes Browser and Unparse and InstantiateTestGraph.

6. **(Add Grapher to classes)** Since we are going to use the Grapher class in this application, double-click on the Classes suitcase in the File Window. In your

Directory Browser, find a copy of Grapher.h in Workshop/Objects and drag its icon to the icon well of the the Classes browser. Note that Grapher appears as a subclass of View.

7. **(Instantiate CustomView as Grapher)** Double-click on the CustomView in MyWindow and the CustomView Inspector will appear. (If only the Project Inspector shows, use the pull-down menu at the top to go to Attributes.) Click on Grapher in the list of View Classes there and then click on OK in the Inspector window. Note that your CustomView is now called Grapher.

8. **(Add Grapher to Project)** At this point you should add Grapher.h to the project. First be sure that .[hm] (class) is highlighted in the Project panel. Then click the add button in the Project Inspector and then find Grapher.m in the Workshop/Objects directory in the Open Panel which appears. Answer *yes* to the question regarding the copying of the file to your project.

9. **(Make connections)** Connect the draw button to testGraphInstance using the **drawIt:** method, then connect the clear button to tthe CustomView directly using the **clear:** action, and connect testGraphInstance to the Grapher customView using the outlet **myGraph**.

10. **(Save Interface and modify code)** Save the graphical interface and add the following boldface code to testGraph.m

```
#import "testGraph.h"
#import "Grapher.h"

@implementation testGraph

- setMyGraph:anObject
{
    myGraph = anObject;
    return self;
}
```

```
- drawit:sender
{
    [myGraph drawAxes :-10.0 :10.0 :-10.0 :10.0];
    [myGraph drawLine :-5.0 :-5.0 :5.0 :5.0];
    return self;
}

@end
```

11. **(Save, compile, and run)** Use **Edit** and **Save** to save this file. Use **Make** to compile and go to the Directory Browser to find and execute testGraph.debug.

12. **(Examine the structure of this application)** Note that our application is itself very simple and that we use the Grapher object to implement all the functionality we usually associate with a graph. We let the Grapher object do all the work of converting our graph scales to pixels on the screen as well as doing the actual drawing itself. To use this object we do not need to know how it is implemented but we need only know how to access its methods. This is the beauty of object-oriented programming. Note also the connections in this application. We have connected the **draw** button to our testGraphInstance with the **drawIt:** method as expected but we have connected the **clear** button directly to the customView using the **clear:** method of the Grapher object itself.

13. **(Examine Grapher object)** Now that we have used the Grapher class, we might be interested in examining its implementation, which is in Grapher.m

```
#import  "Grapher.h"
#import <dpsclient/wraps.h>
#import <appkit/Bitmap.h>

@implementation grapher

+ newFrame:( NXRect *)frm
{
    self = [super newFrame:frm];
    viewWidth = frm->size.width ;
```

```
    viewHeight = frm->size.height ;
    offWindow = [Bitmap newSize:viewWidth  :viewHeight
                        type:NX_UNIQUEBITMAP];
    [offWindow setFlip:NO];

    return self;
}

- drawSelf:(NXRect *)rects :(int)rectCount
{
    PSsetgray(1.0);
    NXRectFill(rects);
    [offWindow composite:NX_COPY toPoint:&rects->origin];

    return self;
}

-drawAxes:(float)minx :(float)maxx :(float)miny :(float)maxy
//Establish scale and draw axes
{
    [offWindow lockFocus];
    PSsetgray(0.0);
    ratiox=viewWidth/(maxx-minx);
    ratioy=viewHeight/(maxy-miny);
    minX=minx;
    minY=miny;
    PSmoveto(-minx*ratiox,0.0);     // draw y-axis
    PSlineto(-minx*ratiox,viewHeight);
    PSstroke();
    PSmoveto(0.0,-miny*ratioy);     //draw x-axis
    PSlineto(viewWidth,-miny*ratioy);
    PSstroke();
    [offWindow unlockFocus];
    [self display];
    return self;
}
```

```
-drawLine:(float)x1 :(float)y1 :(float)x2 :(float)y2
{
    [offWindow lockFocus];
    PSsetgray(0.0);
    PSmoveto((x1-minX)*ratiox,(y1-minY)*ratioy);
    PSlineto((x2-minX)*ratiox,(y2-minY)*ratioy);
    PSstroke();
    [offWindow unlockFocus];
    [self display];
    return self;
}

-drawPoint:(float)x1 :(float)y1
{
    [offWindow lockFocus];
    PSsetgray(0.0);
    PSmoveto((x1-minX*ratiox,(y1-minY)*ratioy);
    PSrlineto(0.0, 0.0);
    PSstroke();
    [offWindow unlockFocus];
    [self display];
    return self;
}

-clear:sender
{
    [offWindow lockFocus];
    PSsetgray(1.0);
    NXRectFill(&bounds);
    [offWindow unlockFocus];
    [self display];
    return self;
}
```

```
@end
```

Comment: The transformations from points (x, y) in the user's graph to pixels (s, t) in the view are given by

```
s = (x-minX)*viewWidth/(maxX-minX)
t = (y-minY)*viewHeight/(maxY-minY)
```

Exercise: Graph the function $y = x^2$ from x=-1 to x=1 by evaluating the function at intervals of 0.1 and then drawing lines between successive points. In C this may be done by

```
for(x1=-1.0;x1<1.0;x1=x1+0.1){
        y1=x1*x1;
        x2=x1+0.1;
        y2=x2*x2;
        [myGraph drawLine:x1 :y1 :x2 :y2];
}
```

Exercise: Draw the same function by graphing 100 points.

Exercise: What methods might one add to Grapher to make it more useful?

Making a Tiff Graphic with Icon (used in DragIt)

1. Open the application **Icon**. If it's not in the **Dock**, look in **NextDeveloper/Demos/Icon.** In Icon go to **"Palettes"** and open all 5 of the Palettes.

2. Create a new Icon file by clicking in **Icon, Window, New.**

3. Now let's create a medium sized sphere graphic that we can use in the **dragit** program.

Click once on the sphere in the **Shapes** palette. A sphere will appear in the **Untitled.tiff** window. Drag it near the top of the window. **Do not** click outside of the sphere, it will disappear! If you do lose it, Select the **marquee** (the dashed square) in the **Tools** palette (in the lower right corner). Drag out a small square and click the sphere in the **Shapes** palette again. To keep the graphic in the window you must **"stamp"** it into the window by clicking on the 3rd icon, top row, of the **Effects** palette.

4. Click the **Coordinates** button in the **Tool** palette. The **first pair** of numbers are the **x, y** coordinates of the the corner of the **marquee**. The **second pair** are the **height and width** of the **marquee**.

(If for some reason you lost the marquee around the sphere, select the **marquee** icon again from the **Tools** palette and drag out another marquee around the sphere. Try to get the marquee lines as close to the sphere as you can without touching it. As you do this make sure that the **right pair** of coordinates in the coordinate box end up being equal.)

Now select the **sizing** icon in the **Effects** palette (2nd box of 2nd row). Place the cross cursor in the lower right corner of the selection marquee with the sphere in it and drag until the **right** pair of coordinates each read **75**. This will reduce the size of the sphere

5. In the **Tool** palette, open the pull down menu at the bottom of the palette and

select **Sover**.

6. While the sphere is still selected (marquee around the sphere) go to **Icon, Window** and do **Save As...** Under the Name field there are 3 large buttons. Click on the left button until it reads **TIFF**. Click on the middle button so that it reads **SAVE SELECTION**. Enter the name you want for the graphic (call it **"sph6"**) and click **OK**. This will save the graphic in a field the same size as the marquee. The width of the marquee will be the approximate diameter of the sphere.

7. We now need to modify the graphic so that the space around the sphere is transparent (if it's not already). We only want the sphere to show when it is displayed in our application, not the white part around the sphere. Go to **Icon, Window** and **Open** the graphic you just saved. The sphere will appear in a small window the same size as the marquee you saved above. You can now close the large window you have been working in if you like. Don't save it.

8. Select the small window with your sphere in it (click it) and Click in **Icon, Detail...** This will place a still smaller window called **Detail** on part of the graphic. Also a window with a magnified detail view of part of the graphic will open. The **Detail** window can be dragged to view different parts of the magnified graphic.

9. Go to the **Shades** palette and click in the rectangle in the **4th column of the top row**. This will select transparent paint that can be used to make the white space around the sphere invisible. (If you want white or a shade of gray choose one of the boxes in the left column. The other columns give degrees of transparence.) If two sliders are showing click the right most column to show the other columns.

10. Now choose the **pencil** in the **Tools** palette. Select the magnified Detail window and drag over a portion of the white area. Now hold the **Command** key and the number **1** key down together. You will see granite sliding by where the white was. This indicates that the region is transparent.

A fast way to paint over an area is to use the **funnel** in the **Tools** palette. Select the **funnel** and move it to the the white area in the **magnified** window. Then click. This will spread transparent paint over the area. Move the **Detail** window to the other corners of the graphic and make them transparent if they're not. Remember,

Command 1 will show which parts of your graphic is transparent.

11. When you have the sphere with all of the surrounding white transparent, do an **Icon, Window, Save As...**

12. If **after** doing the **Save As...** you get another alert box asking whether you want to save any changes, click **NO !! If you click save, the you may delete the changes you saved previously!**

This finishes the construction of the sphere graphic. Quit Icon.

Events and Bezier Curves

In many applications it is desirable to be able to detect user-generated actions such as clicking the mouse or moving the mouse or pressing a key. These actions are called **events** and can be incorporated into applications. The Window Server, which is part of NextStep, uses mouse-down events to make windows active and mouse-dragged events to drag windows. Every time we click on a menu button or highlght a file in the Browser we are creating an event tc which the Window Server responds with some action. When we are typing letters into an editor or word processor, we are creating key-down events. Methods for responding to events of this kind are already written to make sure that the actions we have described will take place. If we want to incorporate events into our own application in some different way, we must override the method which handles that event. In the following example we rewrite the mouseDown: method so it does what we want it to in this application.

Launch **Workshop/Programs/Bezier1** and click the mouse while the cursor is at different points in the view. At each click, a small square is drawn at the cursor position. After 4 clicks a curve is drawn from the first to the fourth point. This is called a cubic Bezier curve andt is determined by 4 points , (x_0,y_0), (x_1,y_1), (x_2,y_2), and (x_3,y_3). The curve passes through (x_0,y_0) and (x_3,y_3) with its slope at (x_0,y_0) defined to be the slope of the line from (x_0,y_0) to (x_1,y_1). similarly, the slope at (x_3,y_3) is the slope of the line from (x_2,y_2).

The two center points are called control points since they control the shape of the resulting Bezier curve by controlling the slope of the curves at the endpoints.

We can control the convexity of the curve by placing the control points at different places..

The advantage of the cubic Bezier curves is that we can store the information necessary to draw this curve by storing only 4 points. To draw the Bezier curve in PostScript we need only give the commands

```
PSmoveto(x0, y0);
PScurveto(x1, y1, x2, y2, x3, y3);
```

In the following example we write our own mouseDown: method which is the method the Window Server will invoke when it detects a mouseDown event while the cursor

is over this view. The method will draw the square at that location, store the coordinates, and draw the Bezier curve if the fourth point has been drawn.

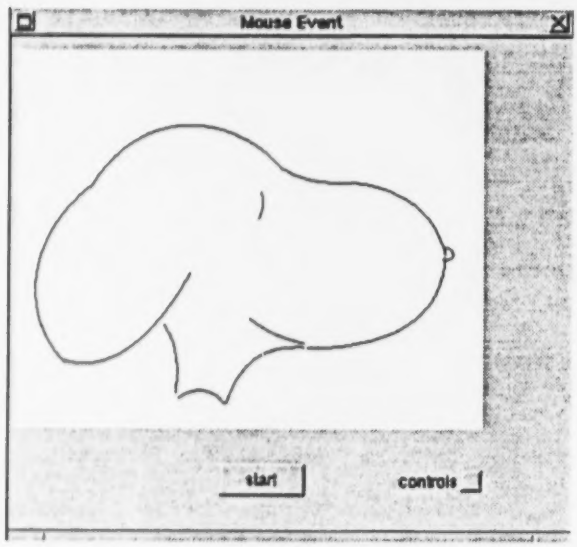
Note that the parameter for the mouseDown: method is a pointer to an event. This pointer, e in this case, will point to the event structure which has information about the event such as the location in the window where the event took place. This point is referred to by e->location in the mouseDown: event below. A complete description of this event structure may be found in Chapter 4 of the Systems Reference Manual.

```
#import "BezierView.h"
#import <dpsclient/wraps.h>

#import <appkit/appkit.h>

@implementation BezierView

+ newFrame:( NXRect *)frm
{
     self = [super newFrame:frm];
     count =0;
     offWindow = [Bitmap newSize:frm->size.width  :frm->size.height
                          type:NX_UNIQUEBITMAP];
     [offWindow setFlip:NO];

     return self;
}

-setControl:anObject
{
    control = anObject;
    return self;
}

- drawSelf:(NXRect *)rects :(int)rectCount
{
     PSsetgray(1.0);
```

```
    NXRectFill(rects);

    [offWindow composite:NX_SATOP toPoint:&rects->origin];

    return self;
}

- mouseDown:(NXEvent *) e
{
   NXPoint pt;

   [self lockFocus];
   pt= e ->location;            //get window coordinates of cursor
   [self convertPoint:&pt fromView:nil];   //change to view coords
   [self unlockFocus];

   [offWindow lockFocus];             //draw to bitmap offwindow

   x[count] = pt.x;                    //store the coordinates
   y[count] = pt.y;
   if (![control state])               //draw control point
        PSrectstroke(x[count]-2.0, y[count]-2.0, 4.0, 4.0);

   if(count==3){                       //draw the cubic Bezier
             PSmoveto(x[count-3], y[count-3]);
             PScurveto(x[count-2], y[count-2],
                  x[count-1], y[count-1],
                  x[count], y[count]);
             PSstroke();
             count = -1;
    }

    [offWindow unlockFocus];
    [self display];
    count++;
```

```
    return self;
}

- clear:sender
{
    [offWindow lockFocus];
    NXEraseRect(&bounds);
    [offWindow unlockFocus];
    [self display];
    count=0;

    return self;
}

@end
```

The main outline of the following figure, with apologies to Charles Schulz, is defined by nine cubic Bezier curves. (Can you guess where the control points are?)



Using the Keydown Event

To introduce the use of the keyDown event into the Bezier application, we will rewrite our Bezier application so it also responds to a keyDown event and writes the character corresponding to that key into the View. Since text is usually written from top to bottom, we will need to change the orientation of the View coordinates. To see how this may be done, examine especially the bold-faced statements of the modified application. This application is in **Workshop/Programs/Bezier2.**

```
#import "BezierView.h"
#import <dpsclient/wraps.h
#import <appkit/appkit.h>

@implementation BezierView

char  str[100];                    //new variables required
float  xMax, yMax;

+ newFrame:( NXRect *)frm
{
    self = [super newFrame:frm];
     count =0;
     offWindow = [Bitmap newSize :frm->size.width
                                 :frm->size.height
                            type:NX_UNIQUEBITMAP];
     xMax=frm->size.width;          //used to center text
     yMax=frm->size.height;         //used to flip y-coordinates
     str[0]='\0';                   //define an empty string
     return self;
}

-setControl:anObject
{
    control = anObject;
    return self;
```

```
}

- drawSelf:(NXRect *)rects :(int)rectCount
{
     PSsetgray(1.0);
     NXRectFill(rects);

     [offWindow composite:NX_SATOP   toPoint:&rects->origin];
     return self;
}


- mouseDown:(NXEvent *) e
{
    NXPoint pt;

    [self lockFocus];
    pt= e->location;
    [self convertPoint:&pt fromView:nil];
    [self unlockFocus];

    [offWindow lockFocus];
    x[count] = pt.x;                  //store the coordinates
    y[count] = yMax-pt.y;             //flip y-coordinate

    if (![control state])            //draw control points
          PSrectstroke(x[count]-2.0, y[count]-2.0, 4.0, 4.0);

    if(count==3){                    //draw the cubic Bezier
        PSmoveto(x[count-3], y[count-3]);
        PScurveto(x[count-2], y[count-2],
                  x[count-1], y[count-1],
                  x[count], y[count]);
        PSstroke();
        count = -1;
```

```
    }
    [offWindow unlockFocus];
    [self display];
    count++;

    return self;
}

 - keyDown:(NXEvent *) ptr    //overwrite given keyDown method
{
    char  str2[2];

    str2[0]=ptr->data.key.charCode;  //store key character
    str2[1]='\0';
    strcat(str, str2);          //concatenate character to str
    [offWindow lockFocus];
    PSmoveto(xMax/2.0-10.0, 10.0);
    PSshow(str);                        //write str to offWindow
    [offWindow unlockFocus];
    [self display];

    return self;
}

- (BOOL) acceptsFirstResponder  //needed for keyDown in View
{
   return YES;
}

- clear:sender
{
    [offWindow lockFocus];
    NXEraseRect(&bounds);
    [offWindow unlockFocus];
    [self display];
    count=0;
```

```
    str[0]='\0';                    //re-initialize str
    return self;
}

@end
```

Note that we have added two methods which will override methods with the same names which are already defined by NextStep. (Note that these two methods are not in the interface file for this application.) Because text coordinates run from top to bottom, we took the line

```
[offWindow setFlip:NO]
```

out of the previous application (the default is YES) and flipped the y-coordinates of the points in our application.

Exercise: Can you rewrite the keyDown method so you can pick the point in the View at which you would like to begin writing text?

A Dial Object

Dials provide an example of something that we may want to use several times in an application, or in several different applications. Objective C encourages user defined objects. This example defines a DialBitmap object and shows its use in a simple program. The dial object could be defined in several different ways, in this case it is defined as a subclass of Bitmap, so the application will need to provide a view to which it can be composited. The view in this case is provided by a CustomView. Values for the dial object would usually be computed by the application, in this simple example they are derived from a slider.

Launch Interface Builder and create a new application. Add a CustomView and a Slider to the Window. Make the CustomView about 200 units square (with the CustomView selected, look at the Miscellaneous menu item in the Inspector Panel to determine the exact size). Save the file to a new directory, calling it dial. Choose project from Interface Builder's menu and click okay in the Inspector Panel which will ask if you wish to create a project in the current directory.

Click on the classes suitcase to open the Class Browser. Create a subclass of View and call it DialView. Choose Attributes from the Inspector panel with the CustomView selected and click DialView and okay.

With DialView in the Classes browser selected, choose class from the Inspector panel. Add an outlet called getSlider, and an action called computeSomething:. Unparse DialView and answer okay to the alert panels that ask if you wish to create DialView.[hm] and add them to the project.

Control-drag a connection from DialView to the slider and select getSlider in the Inspector's class panel. Control-drag a connection from the slider to DialView and select computeSomething:.

Choose Project in the Inspector Panel and click Add... Find a copy of DialBitmap.m (it is in the Objects directory) and click okay to the alert panel to copy it to the project directory.

To make the necessary changes to DialView, select project in the Inspector, and

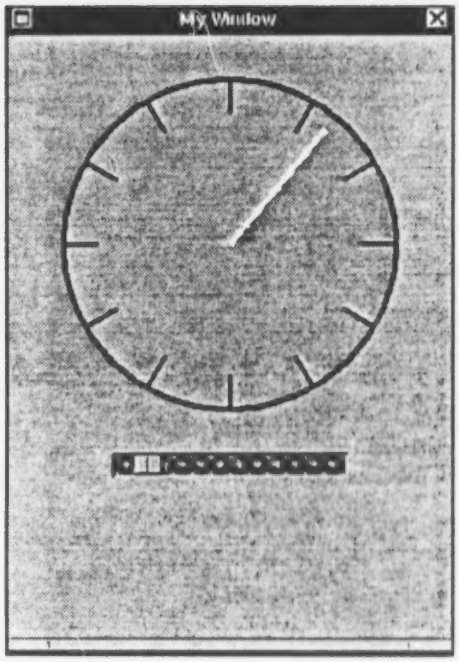
select DialView in the Inspector's browser, then click the open button to launch the editor. The code for DialView.h and DialView.m should look like this - you should add the code which is shown here in bold face.

```
/* Generated by Interface Builder */

#import <appkit/View.h>

@interface DialView:View
{
    id   getSlider;
    id thisDial; // declare an instance variable for DialBitmap
}

- setGetSlider:anObject;
- computeSomething:sender;

@end
```

```
/* Generated by Interface Builder */

#import "DialView.h"
#import "DialBitmap.h"
#import <appkit/appkit.h>

@implementation DialView

+ newFrame : (NXRect *) frm
{
float radius;
float numTicks = 12.0;
```

```
    self = [super newFrame : frm];
    radius = frm->size.width / 2.0;
    thisDial = [DialBitmap new];
    [thisDial makeDial : radius : (int) numTicks];
    return self;
}


- setGetSlider:anObject
{
     getSlider = anObject;
     return self;
}

- computeSomething:sender
{
float angle;

    angle = [getSlider floatValue];
    angle *= 360.0;
    [self lockFocus];
    [thisDial drawDial : angle];
    [self unlockFocus];

     return self;
}


@end
```

Make the project by choosing make from Interface Builder's menu. When the project compiles and links successfully, find the icon for the application, called dial.debug in the Workspace's directory browser and double-click it to launch it.



Your application should look something like this.

How much work would you have to do in order to change the size of the dial?

What would you need to do to include an extra dial in your application?

A Dial controlled by the Animator

This example builds on the previous one to show animation. Often our applications may want to show the behaviour of objects over a period of time in response to computation, or user interaction. To get smooth motion, you need to display images rapidly (video uses 30 frames per second). The time it takes to compute an image may be much longer that a thirtieth of a second, so you should try to pre-compute the parts of the image that do not change much and store them as bitmaps which can be copied onto a view very rapidly.

The Animator object, written by Richard Crandall at NeXT, is a timer object that can be used to control an animation. It messages an action method of your choice which represents one step or computation cycle in the animation; it says how often you want that step to be executed (under ideal circumstances), and it does its best to keep the animation happening in real time when circumstances are not ideal. The Animator object is not part of the Appkit, you can find a copy of it in **NextDeveloper/Examples/Stopwatch**.The Animator object is not the only way to control an animation, the SoundKit provides an object called the Conductor which can be used to time events other than music.

This example uses all the code from the previous example and adds an instance of the Animator object. The method which the Animator controls simply computes a random value for the dial's needle which is slightly different from its previous value.

Open the nib file for the dial object in the previous example by double clicking on its icon.

Click Add in the Project Inspector panel, and find a copy of the Animator object to add to the project. It is in **NextDeveloper/Examples/Stopwatch**. ClickOkay to the alert panel which asks if you want to add a copy of it to the project directory.

Select the DialView.[hm] in the Inspector panel and click open. Add the code in bold face below.

```
/* Generated by Interface Builder */

#import <appkit/View.h>

@interface DialView:View
{
    id   getSlider;
    id thisDial;
    id timer;
    float computedValue;
}

- setGetSlider:anObject;
- computeSomething:sender;

@end


#import "DialView.h"
#import "DialBitmap.h"
#import <appkit/appkit.h>
#import "Animator.h"

// some definitions for the random number generator
extern void srandom();
extern random();
#define RANDINT(n)  (random () % ((n)+1))
#define INITRAND srandom (time (0))


@implementation DialView

+ newFrame : (NXRect *) frm
{
float radius;
```

```
float numTicks = 12.0;

    self = [super newFrame : frm];
    radius = frm->size.width / 2.0;
    thisDial = [DialBitmap new];
    [thisDial makeDial : radius : (int) numTicks];

    INITRAND;
    // initialize the Animator
    timer = [Animator newChronon : 0.0
              adaptation : 0.0
              target : self
              action : @selector (step:)
              autoStart : NO
              eventMask : NX_ALLEVENTS];
    // and start it immediately
    [timer startEntry];
    return self;
}


- setGetSlider:anObject
{
     getSlider = anObject;
     return self;
}

- computeSomething:sender
{
float angle;

    angle = [getSlider floatValue];
    angle *= 360.0;
    [self lockFocus];
    [thisDial drawDial : angle];
    [self unlockFocus];
```

```
    return self;
}

-step : sender
{

   [self lockFocus];
   computedValue +=  (float) (RANDINT (10) - 5);
   [thisDial drawDial : computedValue];
   [self unlockFocus];
   [self display];
   return self;
}

@end
```

Save the changes to the h and m files, and choose Make from Interface Builder's menu. When the program compiles successfully, find it's icon (dial.debug) in the dial directory that you created for the project, and double click to launch it.

What would you need to do to change this program into a clock?

Look at **gravity** and **tetris** for other examples of the use of the Animator.

DragIt

You now have a choice! To use **Interface Builder** to construct the **dragIt** program or If you have had enough of **Interface Builder** for now, use **WSDragIt!**

WSDragIt contains a nearly complete interface for **dragIt**. The only thing you will need to do is (1) add your **Tiff** graphic file to the project and (2) copy the mouseDown method to the **.m** file.

If you want to build dragit yourself, skip down to **Building dragIt from Scratch**.

Building dragIt the Easy Way

1. Find the file **WSDragIt** and double click on **IB.proj**. This will start IB.

2. In **Inspector, Project** click on **.tiff** in the **Type** panel. Then click **Add**. In the **Open** box find and select the **Tiff** file you created in **Icon** and click **OK**. Answer **Yes** to any question asked.

3. You must now let **IB** know about the **Tiff** file addition! Do **Inspector, Class** to open the **Classes** window. Find and select **myView** (which is a sub class made for **View**).

4. Open the **Operations** pull down menu in the **Classes in dragIt** window and select **Parse**. Do **NOT** select **Unparse!!** It will delete the code prepared for WSDragit! Answer **Yes** to any question asked.

5. Now we must add the **mouseDown** method to the **dragIt.m** file. Go to **Inspector, Project** and select **.[hm] (class)** in the **Type** panel. Then double click on myView.[hm] in the right hand panel. This will open the **dragIt .h** and **.m** files.

6. Now click in the **WSDragIt** file window (which you open at the start of this section). Find the file **MouseDown method** and open it (double click on it).

7. Copy the whole file by doing **Command a** then **Command c**. Go back to the **dragIt.m** file opened in **step 5**. Past the **mouseDown** method at the end of the **dragIt.m** file just before **@end.** Do the past with **Command v**.

8. Save the **dragIt.m** file. You can save with **Command s** if you wish.

9. Close **dragIt.m** and **dragIt.h** and get back into **IB** (by clicking the **Inspector** or in the **MyWindow** of **IB**.

10. We can now compile the **dragIt** program. Do **IB, File, Make**.

11. Now run the **dragIt**! Go to the **WSDragIt** file, **Update** it (with **Command u**), find the **dragIt.debug** icon and double click on it. That's it, you are finished!

You will see a ball in the window of your program which you can click and drag around. If you click on the Smear On / Off button, the ball will leave a trail as it's dragged!

Building dragIt from Scratch

1. Go to **IB** and do **File, New Application**

2. Do **IB, File, Save As...**, then go to **me/Apps/YourDirectoryName** and create a sub-directory named **MyDragIt** with a file in it called **dragIt.** To do this enter **MyDragIt/dragIt** after you have selected **YourDirectoryName.** Click **OK**. If an attention box comes up click **Create**. The name of the application will be **dragIt**.

3. Go to **IB, File, Project...** If this is a new project a panel will come up saying that there is no project. If so, click **OK** and one will be created.

4. From the **Palettes** window drag a **CustomView** into the **My Window** window. Size it so that it nearly fills the right 2/3's of the window.

5. Drag a **Button** from **Palettes** to **My Window** and place it left of **CustomView**. Rename the **Button** to **Smear On / Off**.

6. Go to the **"Inspector"** (If you lost the **Inspector** window go to **IB, Window, Inspector**.). Go to the pull down panel near the top of the **Inspector** window and click **Class**. This will bring up another window called **Classes in dragIt**. (Remember that the name of this application we are building is **dragIt**, the name we gave it when we created and saved the directory **MyDragIt**.)

7. In the window **Classes in dragIt**, find **View** and click on it. Open the pull down menu **Operations** and click on **Subclass**. Now double click on **Subclass 1** which appears in the field below the **.h** icon and rename it **myView**. Press return.

8. Go to the **My Window** window and click in the **CustomView** area.

9. Now go to the **Inspector, Attributes** and find **myView** in the panel and click on it. Then click **OK**. This will rename **CustomView** to **myView**.

10. Click in the **Classes in dragIt** window and make sure **myView** is selected. Then go to the **Inspector, Attributes** and select **Action** from the button above the field near the bottom of the **Inspector** window. Type **smear** in the field and press return. This will add smear to the **Action** panel.

11. Hold the **"Control"** key down on the keyboard and drag a connection from the **Button** to **myView** in the **My Window** window. Select **"smear"** from the **Actions** panel and click on **Connect**.

12. Go back to the **Classes in dragIt** window, open the **Operations** pull down menu, and select **Unparse**. Click **OK** or **Yes** to any alert panels that come up.

13. Now we must include the **Tiff** file graphic that we want to display when the application runs. Go to **Inspector, Project** and select **.tiff** in the left panel called **"Type"**. Click **Add**. This will bring up the **Open** window. Find and select the **sph6.tiff** file you saved above and click **OK**. If the **Tiff** file you want is in some other directory, find it and select it. An alert panel will come up asking if you want to make a copy for your directory. Click **Yes**.

14. Go to **IB**, **File** and **Save**.

We are now finished with the Interface Builder! All we need to do now is add the necessary code to the .h and .m files made by Interface Builder.

We now have two choices, write our code while still in **Interface Builder** or leave **Interface Builder** and write the code from the **Shell**.

Some find it more convenient and less cluttered to do things from the **Workspace** and **Shell**. You my find otherwise. Try both and take your choice.

Let's do things from the Shell!

1. Quit the **Interface Builder** and open the **MyDragIt** directory. Also open the **Shell** (If it's not in the **Doc** it's in **NextApps/**).

2. Use **UNIX** commands to get into the **MyDragIt** directory (example: **cd Apps/YourDirectoryName/MyDragIt**).

3. In the Workspace **MyDragIt** directory you will see about a dozen files, two of which will be **myView.h** and **myView.m**. Double click on **myView.m. Edit** will open and display the code made by **Interface Builder**.

4. Above last line "**@end**" and the "**}**" (which marks the end of the switch sender code) change so that it matches the code called **myView.m Code** given below. **Do this using cut and past!** Also add the code

```
#import <appkit/appkit.h>
```

after the line **#import "myView.h"**, which was created by **Interface Builder**.

5. Change the **myView.h** file so that it matches the **myView.h Code** listing given below.

6. Go to the **Shell** window (remember you should be in the **MyDragIt** directory) and type **make** and press return. Your program will then be compiled. If all goes well you will get no error messages (except perhaps a warning - ignore it) and the UNIX prompt.

7. To run your program type its name. In our case remember its name is **dragIt**. The program can also be run by double clicking on the **Icon** displayed in your directory. It may be necessary to update the directory window to show its most recent contents. To do this go to **Workspace, Window, Update** (or do a **Command u**).

You will see a ball in the window of your program which you can click and drag around. If you click on the Smear On / Off it button, the ball will leave a trail as it's dragged!

Code Section

dragit.h Code The additions are shown in **bold**.

```
/* Generated by Interface Builder */

#import <appkit/View.h>

@interface myView:View
{
   id graphic;
   int Flag;          //Flag for smearing
   float Rd;          //Radius of graphic
   float xpos,ypos,xcenter,ycenter;
   //Position of graphic and
   //center of myView.
}

- smear:sender;
```

```
@end
```

dragit.m Code The additions are shown in **bold**.

```
/* Generated by Interface Builder */

#import "myView.h"
#import <appkit/appkit.h>

@implementation myView

+ newFrame:(NXRect *)rect
{
   NXSize gg;

   self = [super newFrame:rect];
   Flag = 0;
   xcenter = rect->size.width/2.0;
         //x center of myView
   ycenter = rect->size.height/2.0;
   //y center of myView
   graphic = [Bitmap findBitmapFor:"sph6.tiff"];
   [graphic getSize:&gg];
   Rd = gg.width/2.0;
   xpos = xcenter;
   ypos = ycenter;

   return self;
}

-drawSelf:(NXRect *)rect :(int)rectCount
{
   NXPoint p;
```

```
    NXRect vu;

    [self getBounds:&vu];
    p.x = (int)(xpos - Rd);
    p.y = (int)(ypos - Rd);
    [self lockFocus];
    if(Flag == 0)
          {
                PSsetgray(NX_DKGRAY);
                NXRectFill(&vu);
          }
    [graphic composite:NX_SATOP toPoint:&p];
    [self unlockFocus];

    return self;
}

- smear:sender
{
    Flag = [sender state];
     return self;
}

//The MouseDown method can go here.

- mouseDown:(NXEvent *)theEvent
    //See Reference Manual: Concept
    //page 7-60.
{
    register int inside;
    int shouldLoop = YES;
    int oldMask,i;
    NXEvent *nextEvent;
    NXRect rr;
    NXPoint p;
    float lxpos,rxpos,bypos,typos;
```

```
//left, right, bottom, and top of
//x and y positions

oldMask = [window addToEventMask:NX_LMOUSEDRAGGEDMASK];
while(shouldLoop)
{
      nextEvent = [NXApp getNextEvent:(NX_MOUSEUPMASK |
            NX_MOUSEDRAGGEDMASK)];
      inside = NXMouseInRect([self convertPoint:&nextEvent->
            location fromView:nil],
            &bounds,
      [self isFlipped]);

switch(nextEvent->type)
{
case NX_LMOUSEUP:
shouldLoop = NO;
break;
      case NX_LMOUSEDRAGGED:
      if(inside)
      {
            lxpos = xpos - Rd;
            //Bounds for draging.
            rxpos = xpos + Rd;
            bypos = ypos - Rd;
            typos = ypos + Rd;
            p = theEvent->location;
            if((p.x>lxpos)&(p.x<rxpos)&(p.y>bypos)&(p.y<typos))
            {
                  xpos = p.x;
                  ypos = p.y;
            }
            [self lockFocus];
            [self display];
            [self unlockFocus];
      }
```

```
            else
            break;
        }
        }
        [window setEventMask:oldMask];

        return(self);
}

@end
```

Using the NeXT Computer for Instructional Applications by The California State University, Los Angeles NeXT User's Group

Following are a series of screen dumps from some of the applications that we have developed. For more information please contact any one of us at Cal State LA.

Don Kiel, Math & Computer Sciences

Gary Novak, Geology

Bob Desharnais, Biology

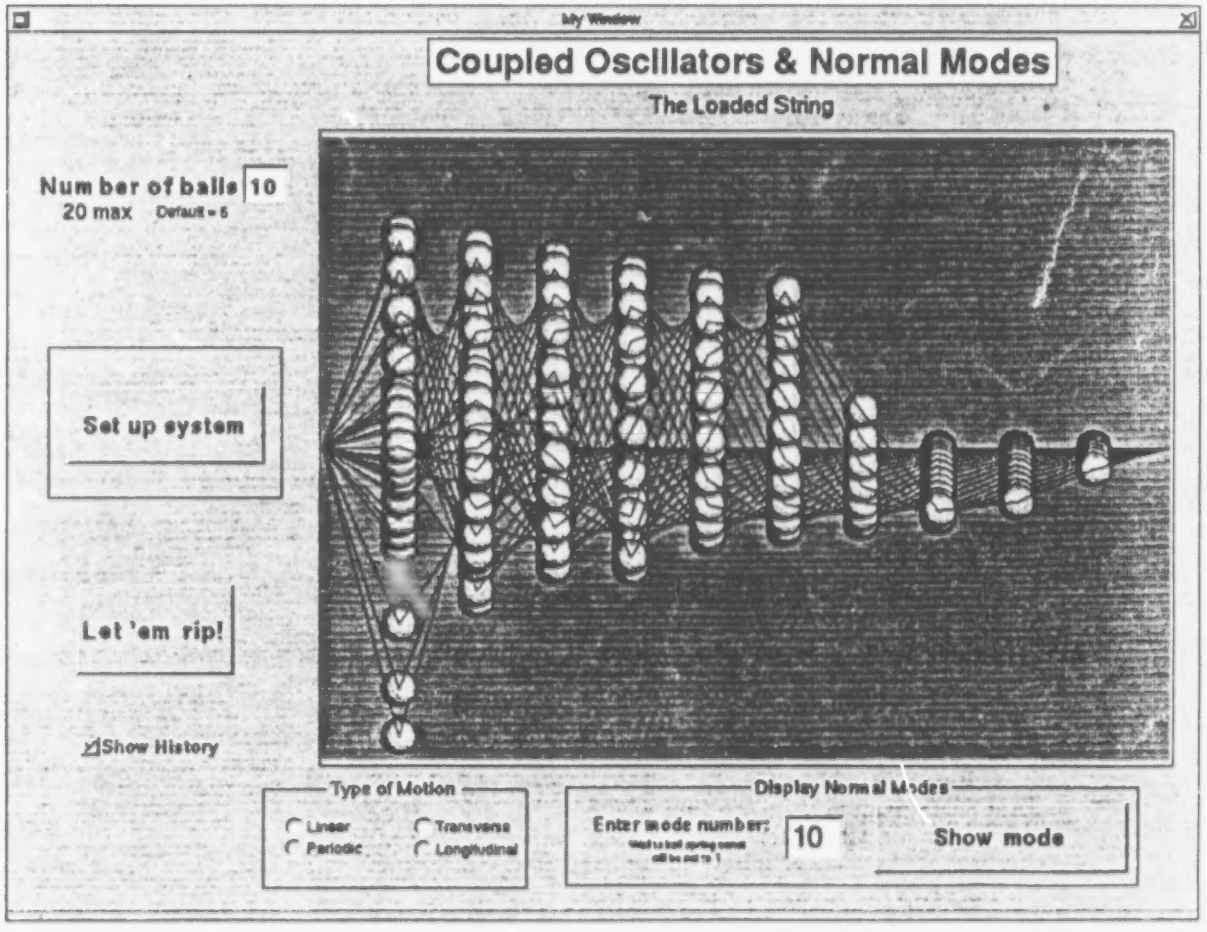
Trina Valencich, Chemistry

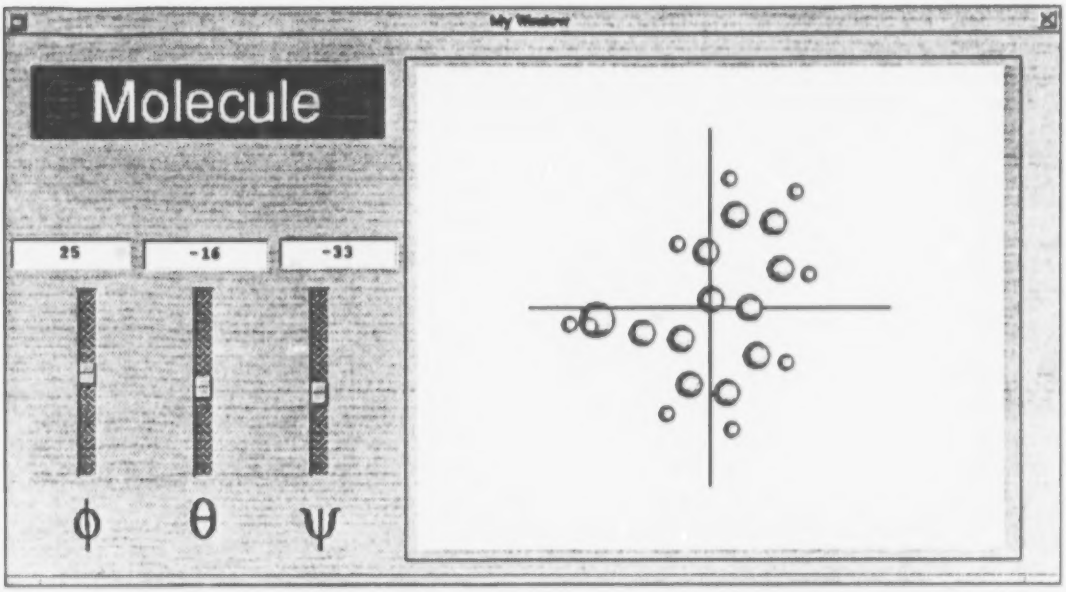
Tony Longson, Fine Arts

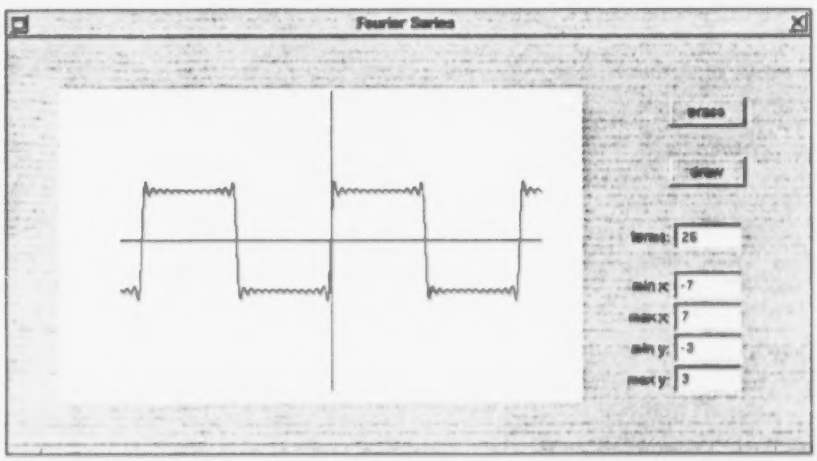
Roland Carpenter, Physics & Astronomy

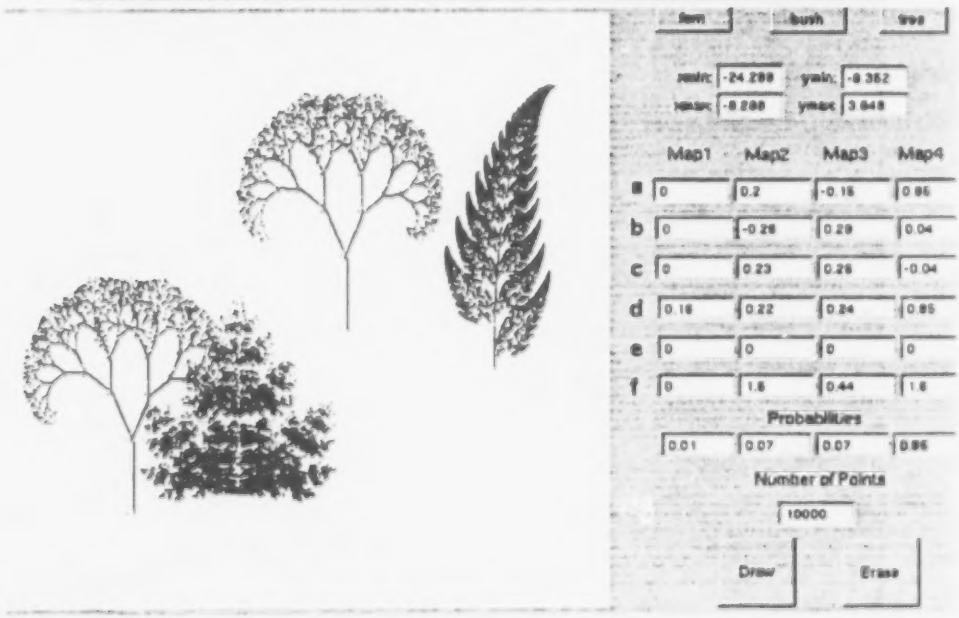
Vickie & Joe Bragin, Chemistry

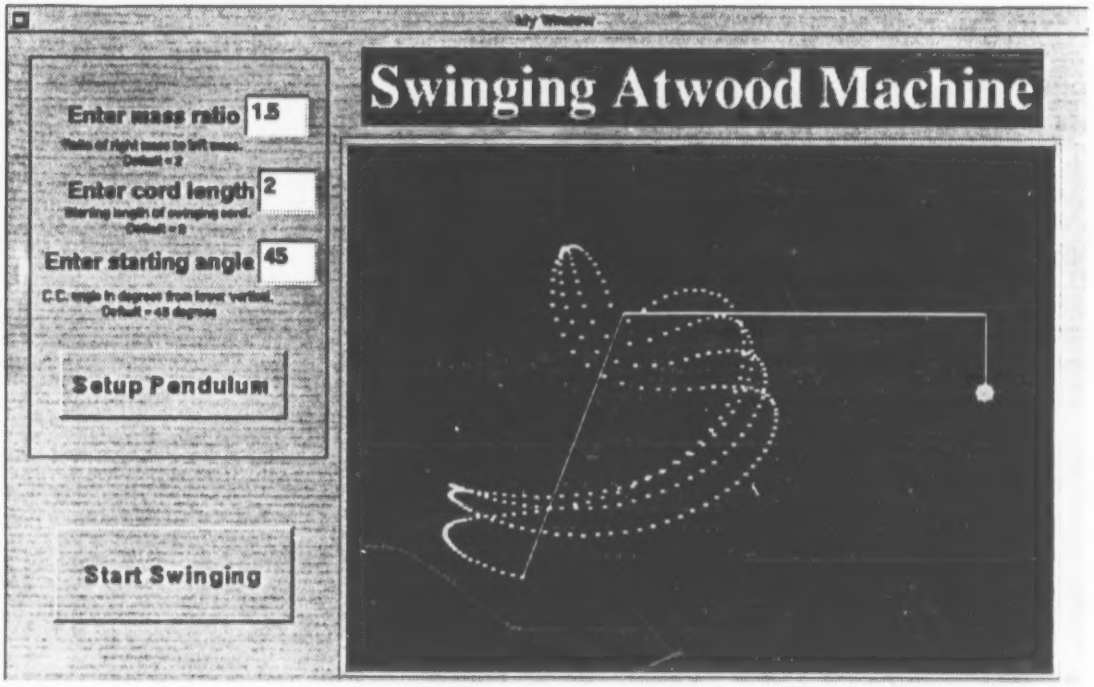
Richard Keys, Chemistry

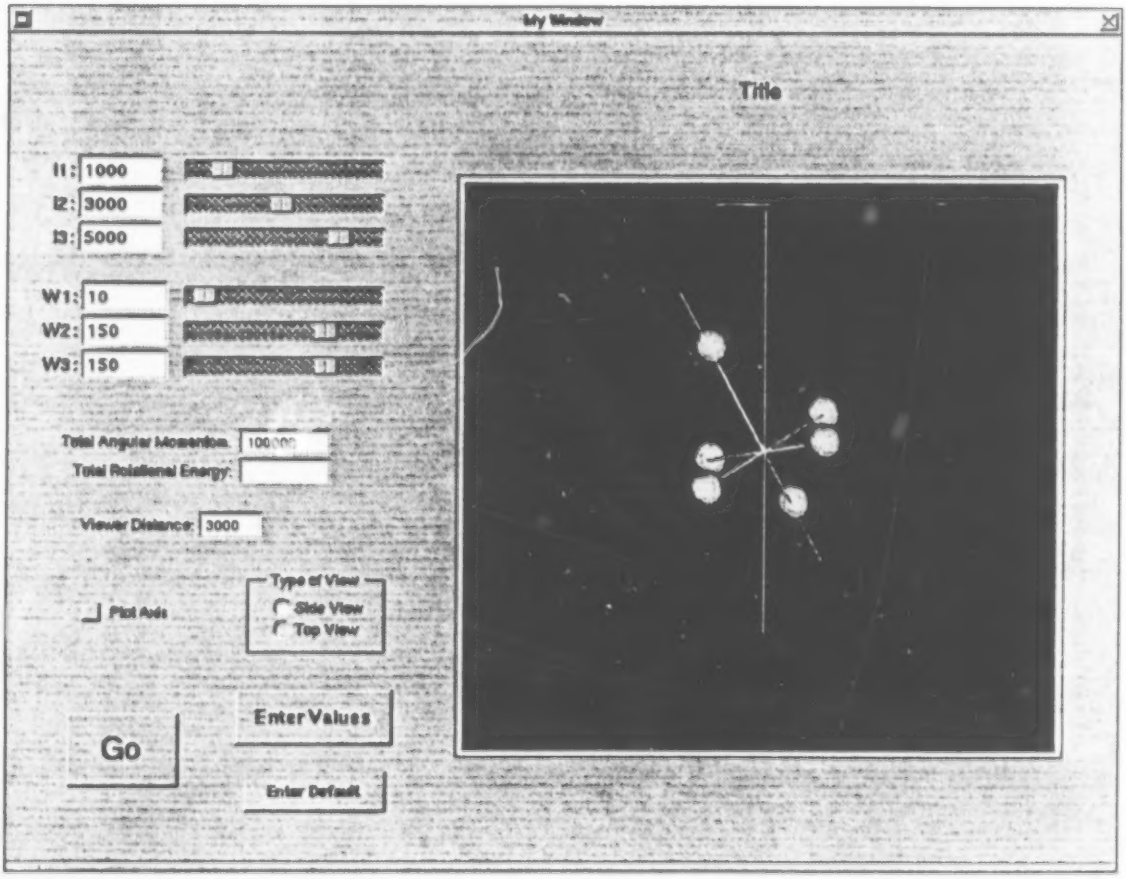



My Window
Molecule
25
-16
-33
ϕ
θ
ψ

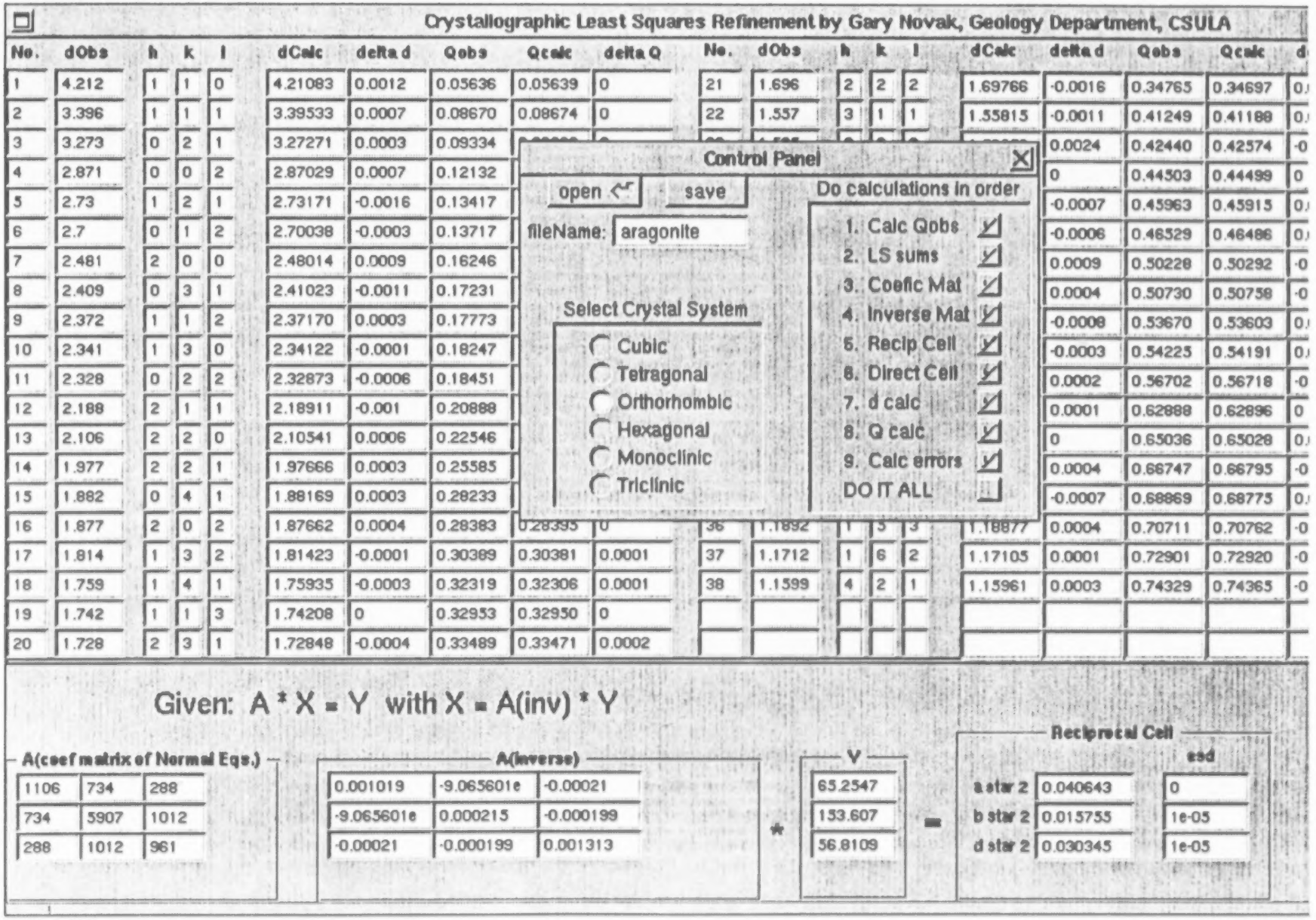

Fourier Series
erase
draw
terms: 25
min x: -7
max x: 7
min y: -3
max y: 3


fern
bush
tree
xmin: -24.299
ymin: -9.352
xmax: 8.268
ymax: 3.848
Map1
Map2
Map3
Map4
a 0 0.2 -0.15 0.85
b 0 -0.26 0.28 0.04
c 0 0.23 0.26 -0.04
d 0.16 0.22 0.24 0.85
e 0 0 0 0
f 0 1.6 0.44 1.6
Probabilities
0.01 0.07 0.07 0.85
Number of Points
10000
Draw
Erase





BEST COPY AVAILABLE



No.	dObs	h	k	l	dCalc	delta d	Qobs	Qcalc	delta Q
1	4.212	1	1	0	4.21083	0.0012	0.05636	0.05639	0
2	3.396	1	1	1	3.39533	0.0007	0.08670	0.08674	0
3	3.273	0	2	1	3.27271	0.0003	0.09334		
4	2.871	0	0	2	2.87029	0.0007	0.12132		
5	2.73	1	2	1	2.73171	-0.0016	0.13417		
6	2.7	0	1	2	2.70038	-0.0003	0.13717		
7	2.481	2	0	0	2.48014	0.0009	0.16246		
8	2.409	0	3	1	2.41023	-0.0011	0.17231		
9	2.372	1	1	2	2.37170	0.0003	0.17773		
10	2.341	1	3	0	2.34122	-0.0001	0.18247		
11	2.328	0	2	2	2.32873	-0.0006	0.18451		
12	2.188	2	1	1	2.18911	-0.001	0.20888		
13	2.106	2	2	0	2.10541	0.0006	0.22546		
14	1.977	2	2	1	1.97666	0.0003	0.25585		
15	1.882	0	4	1	1.88169	0.0003	0.28233		
16	1.877	2	0	2	1.87662	0.0004	0.28383	0.28395	0
17	1.814	1	3	2	1.81423	-0.0001	0.30389	0.30381	0.0001
18	1.759	1	4	1	1.75935	-0.0003	0.32319	0.32306	0.0001
19	1.742	1	1	3	1.74208	0	0.32953	0.32950	0
20	1.728	2	3	1	1.72848	-0.0004	0.33489	0.33471	0.0002

No.	dObs	h	k	l	dCalc	delta d	Qobs	Qcalc	d
21	1.696	2	2	2	1.69766	-0.0016	0.34765	0.34697	0.0
22	1.557	3	1	1	1.55815	-0.0011	0.41249	0.41188	0.0
						0.0024	0.42440	0.42574	-0
						0	0.44503	0.44499	0
						-0.0007	0.45963	0.45915	0.0
						-0.0006	0.46529	0.46486	0.0
						0.0009	0.50228	0.50292	-0
						0.0004	0.50730	0.50758	-0
						-0.0008	0.53670	0.53603	0.0
						-0.0003	0.54225	0.54191	0.0
						0.0002	0.56702	0.56718	-0
						0.0001	0.62888	0.62896	0
						0	0.65036	0.65028	0.0
						0.0004	0.66747	0.66795	-0
						-0.0007	0.68869	0.68775	0.0
36	1.1892	1	5	3	1.18877	0.0004	0.70711	0.70762	-0
37	1.1712	1	6	2	1.17105	0.0001	0.72901	0.72920	-0
38	1.1599	4	2	1	1.15961	0.0003	0.74329	0.74365	-0

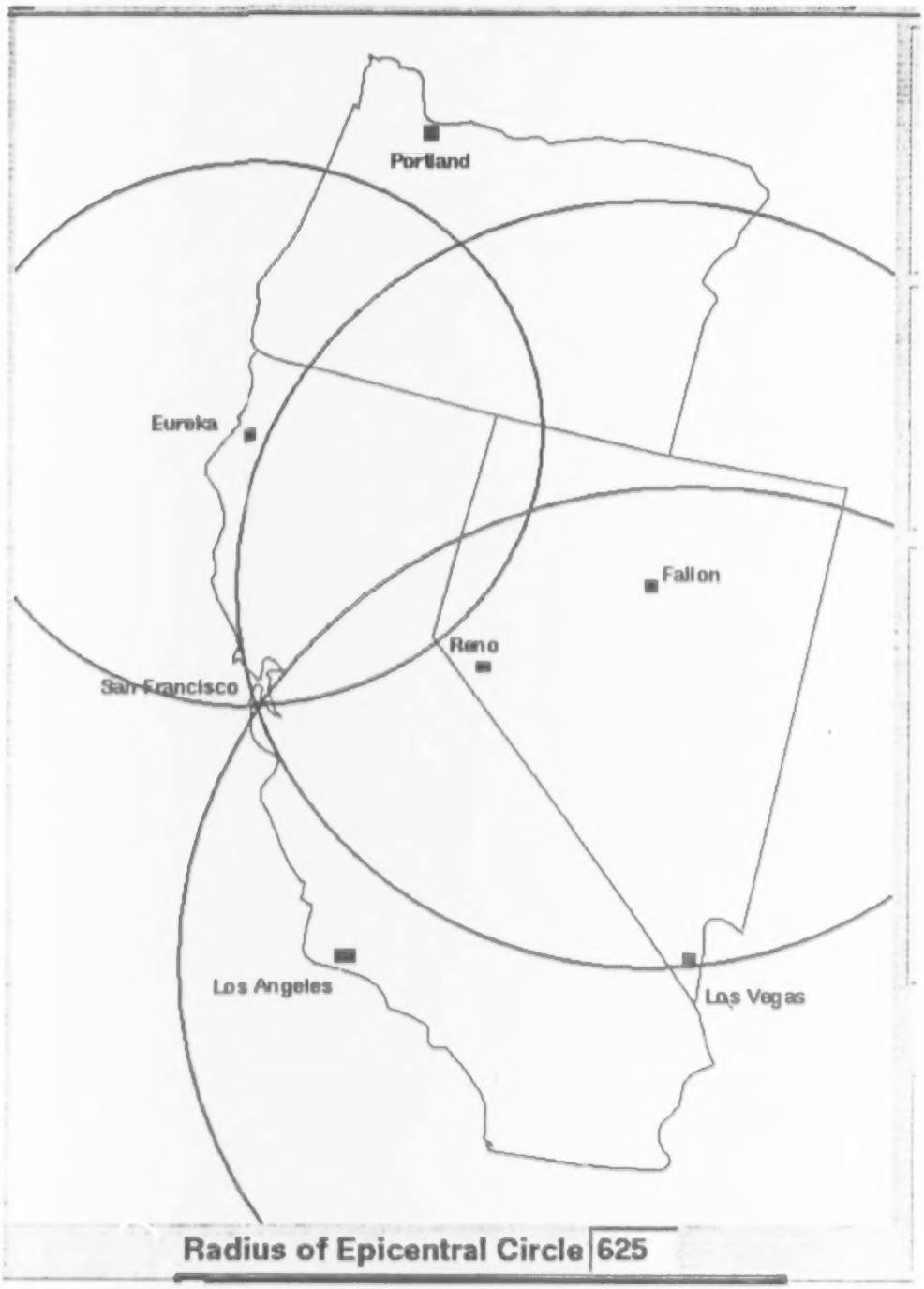
A(coef matrix of Normal Eqs.)		
1106	734	288
734	5907	1012
288	1012	961

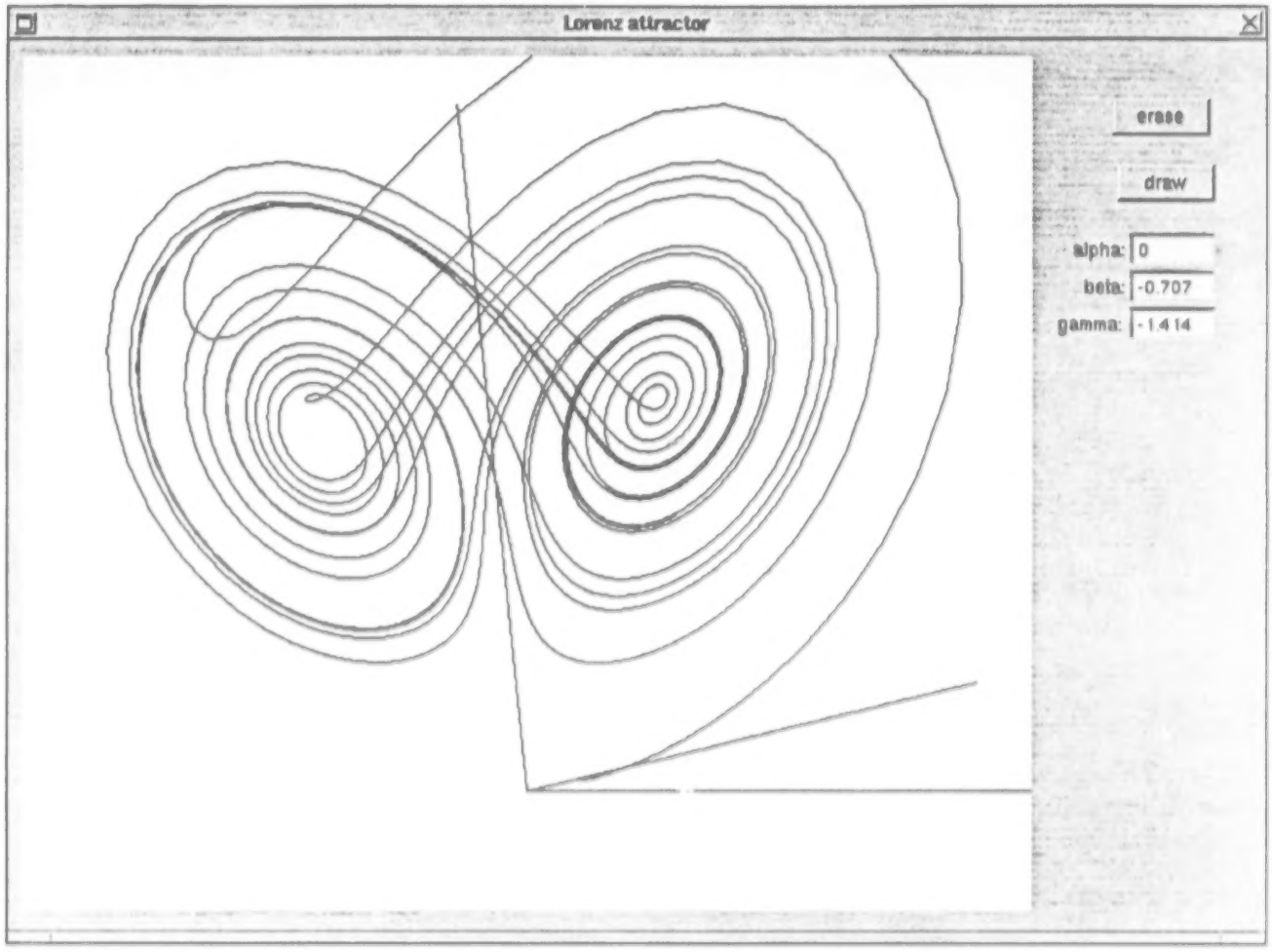
A(inverse)		
0.001019	-9.065601e	-0.00021
-9.065601e	0.000215	-0.000199
-0.00021	-0.000199	0.001313

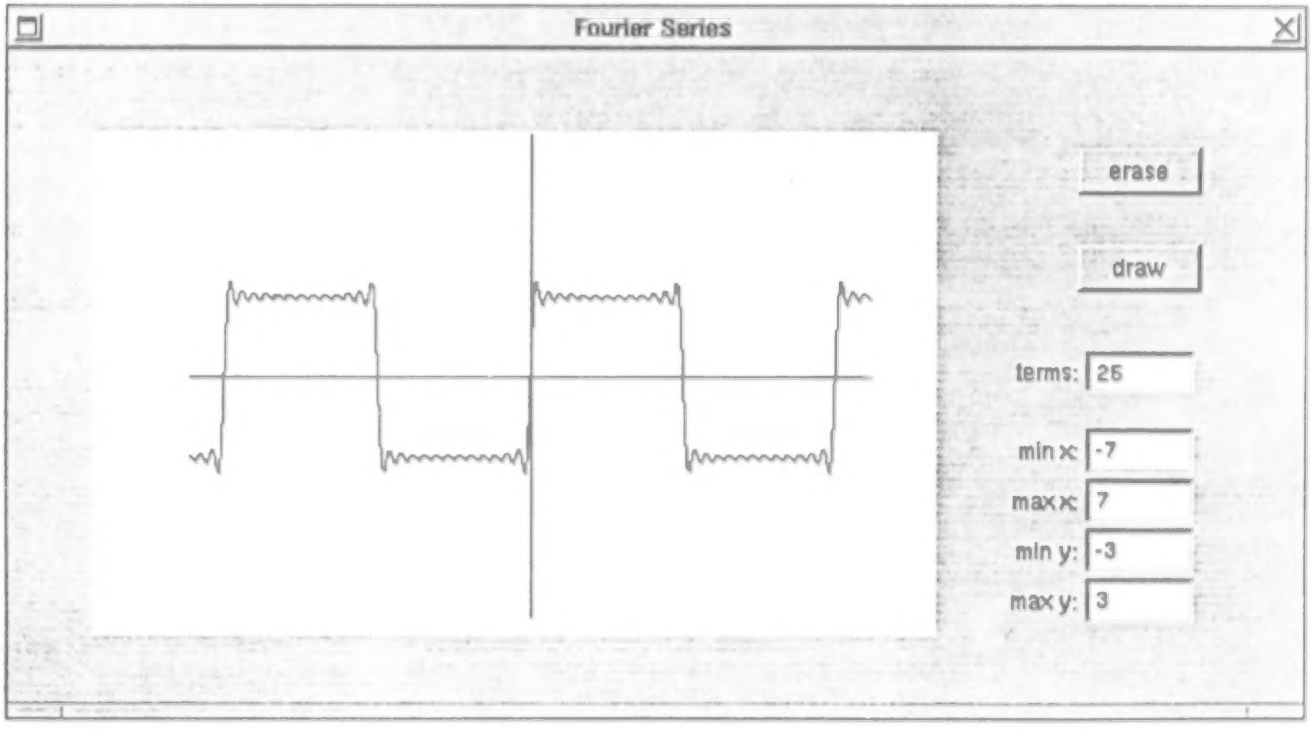
Y
65.2547
133.607
56.8109

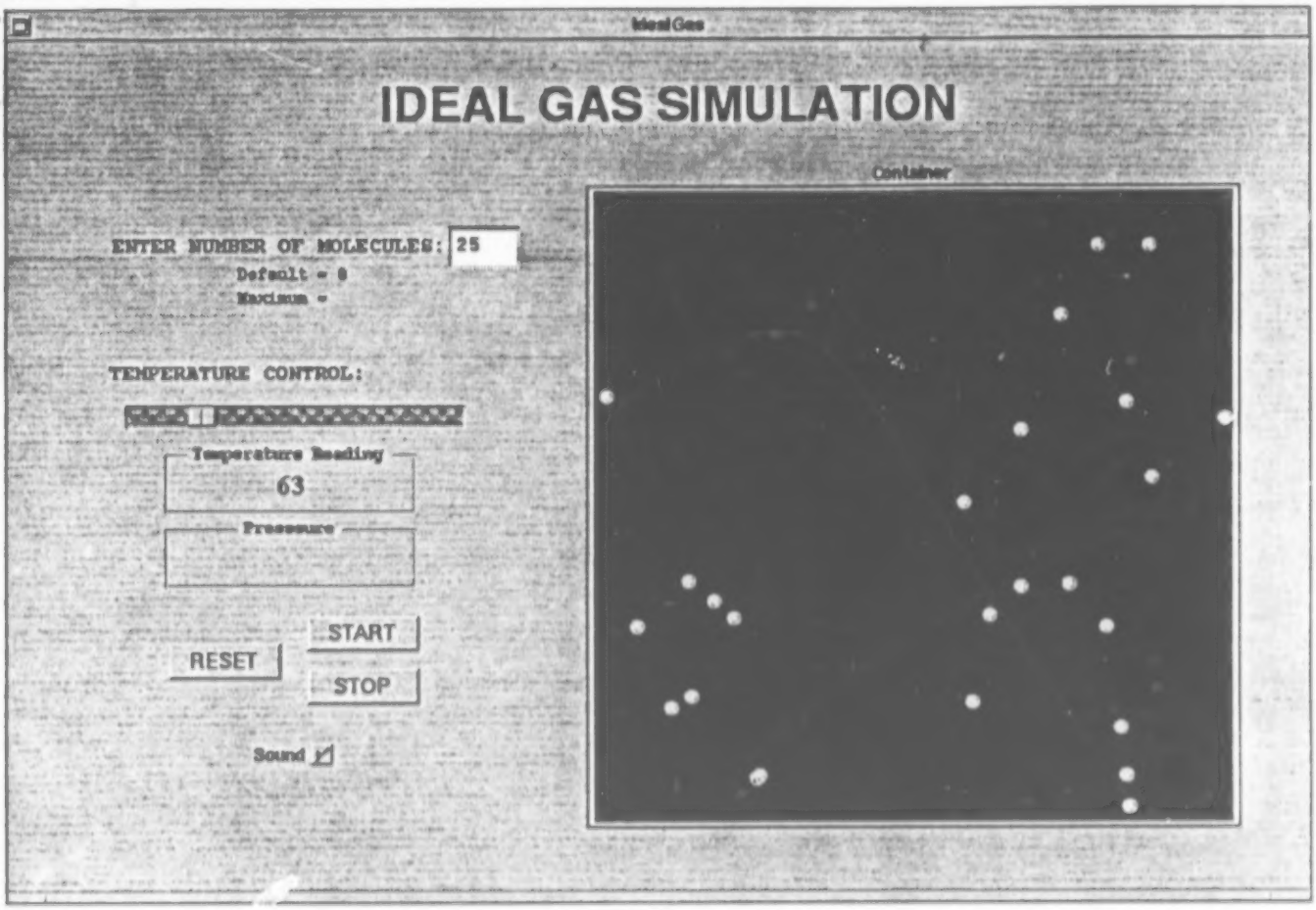
* =

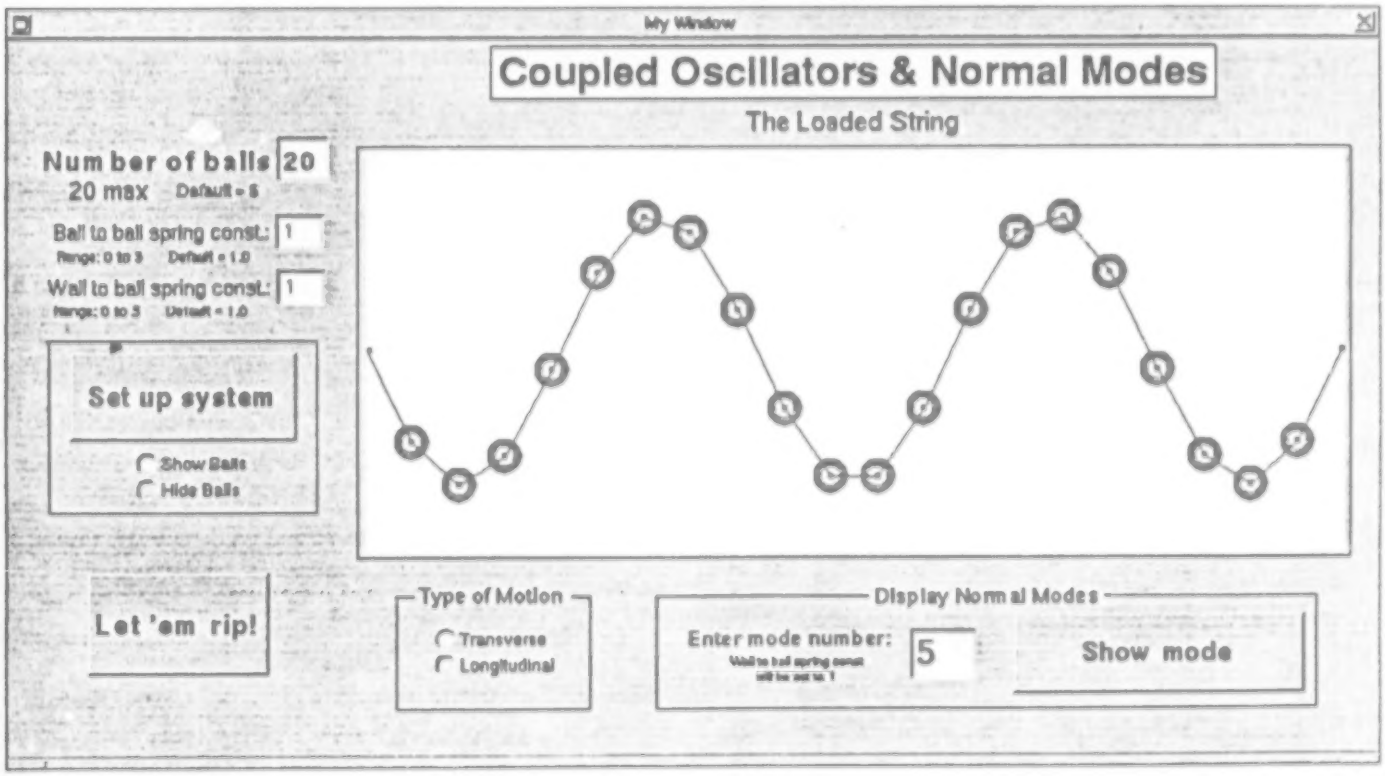
Reciprocal Cell		esd
a star 2	0.040643	0
b star 2	0.015755	1e-05
d star 2	0.030345	1e-05


Portland
Eureka
Fallon
Reno
San Francisco
Los Angeles
Las Vegas
Radius of Epicentral Circle 625

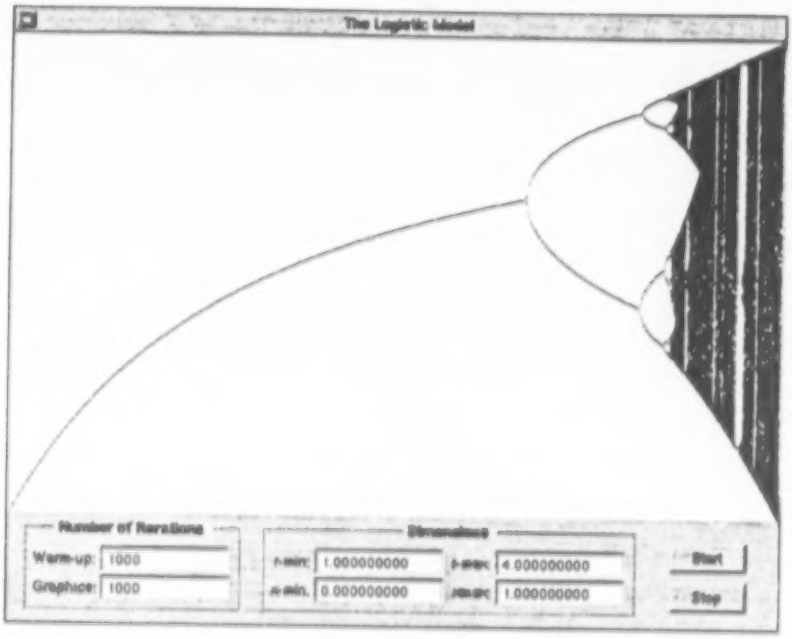

Lorenz attractor
erase
draw
alpha: 0
beta: -0.707
gamma: -1.414


Fourier Series
erase
draw
terms: 25
min x: -7
max x: 7
min y: -3
max y: 3

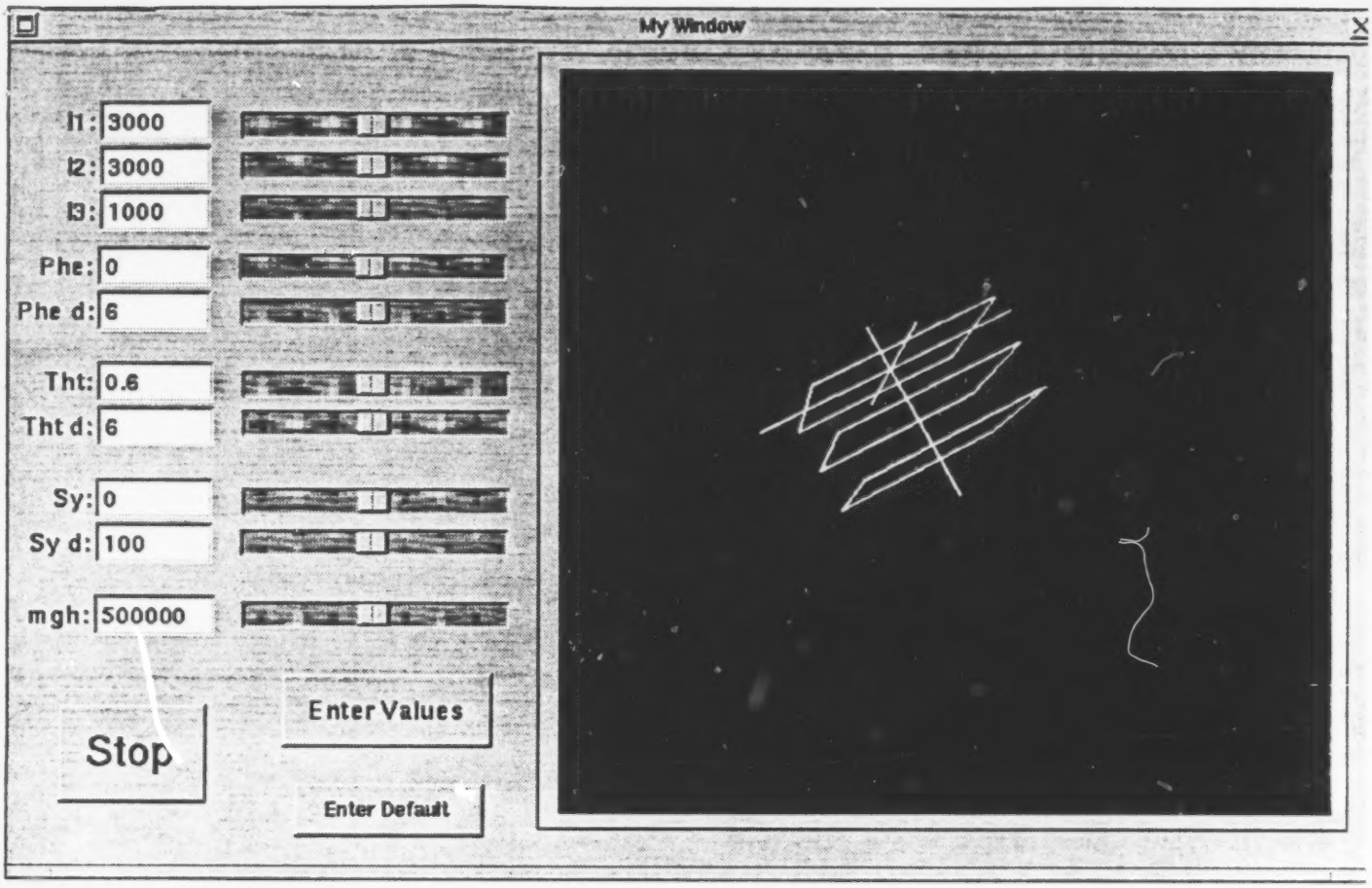




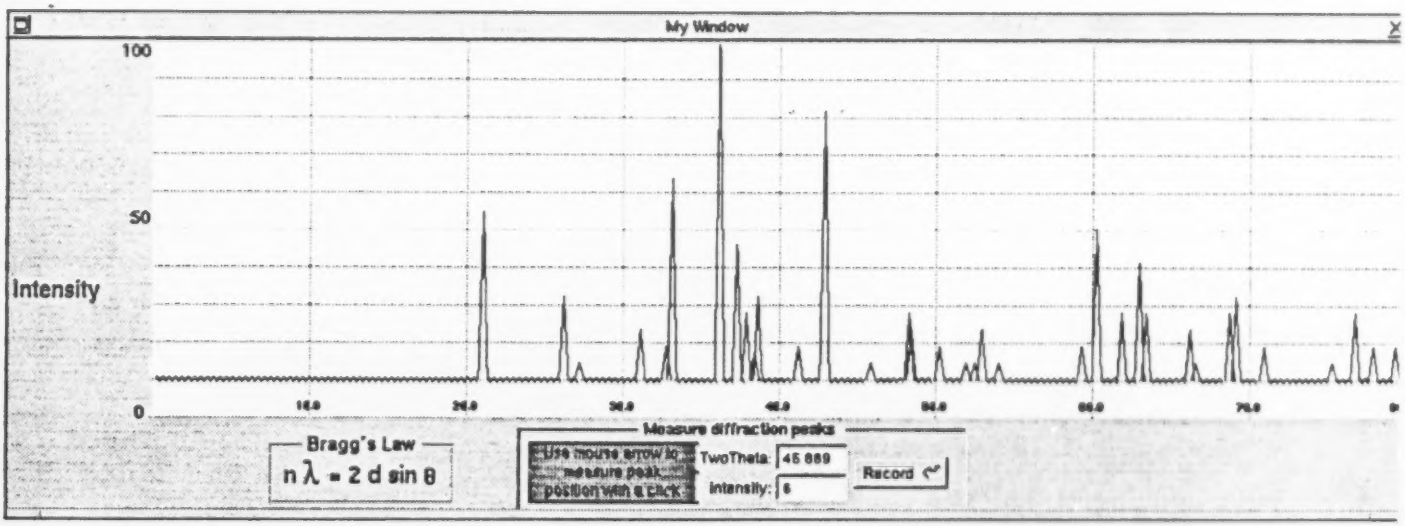
BEST COPY AVAILABLE



BEST COPY AVAILABLE



Application to illustrate rotation & precession of an object about a point

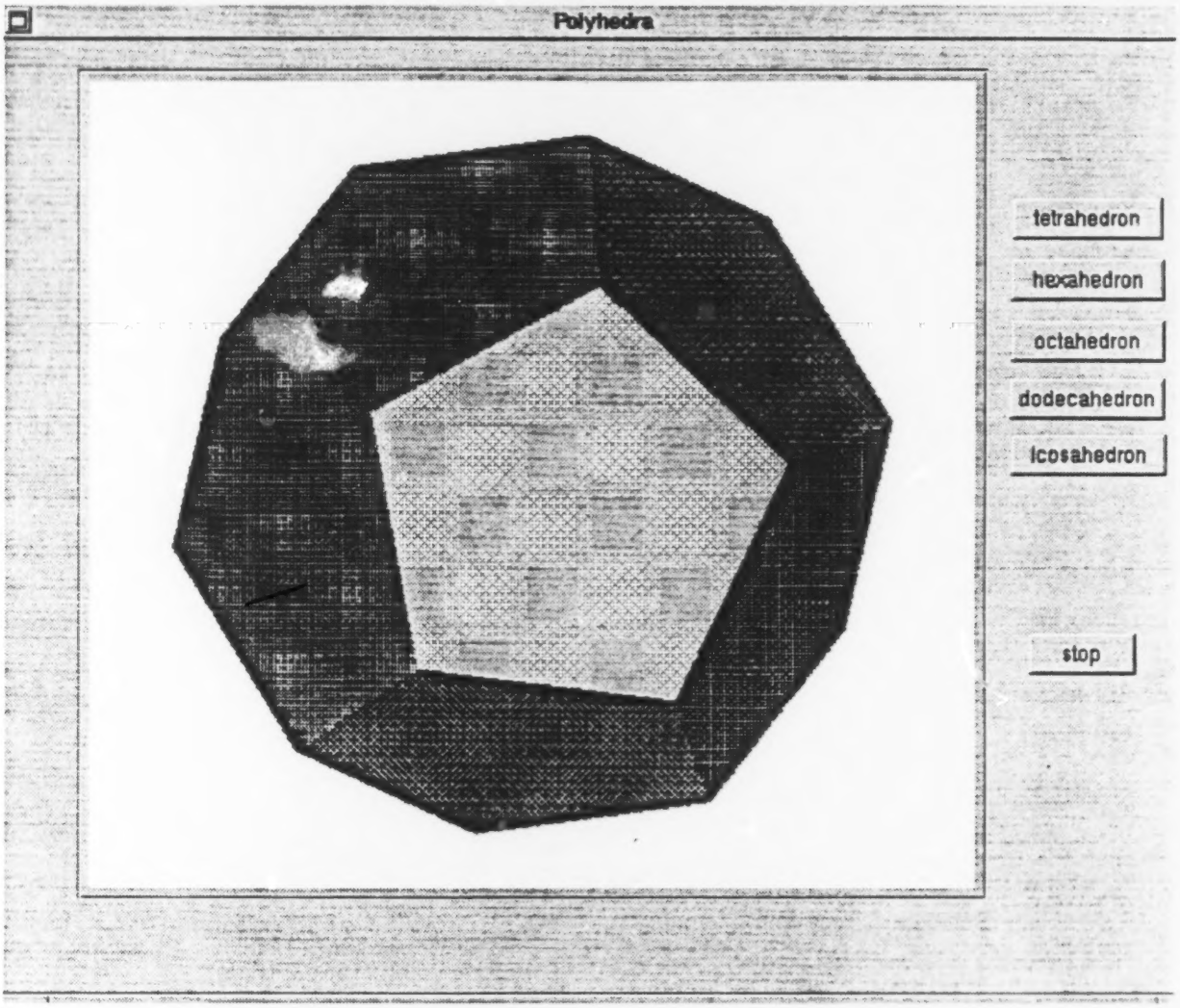


X-ray diffraction strip chart

BEST COPY AVAILABLE

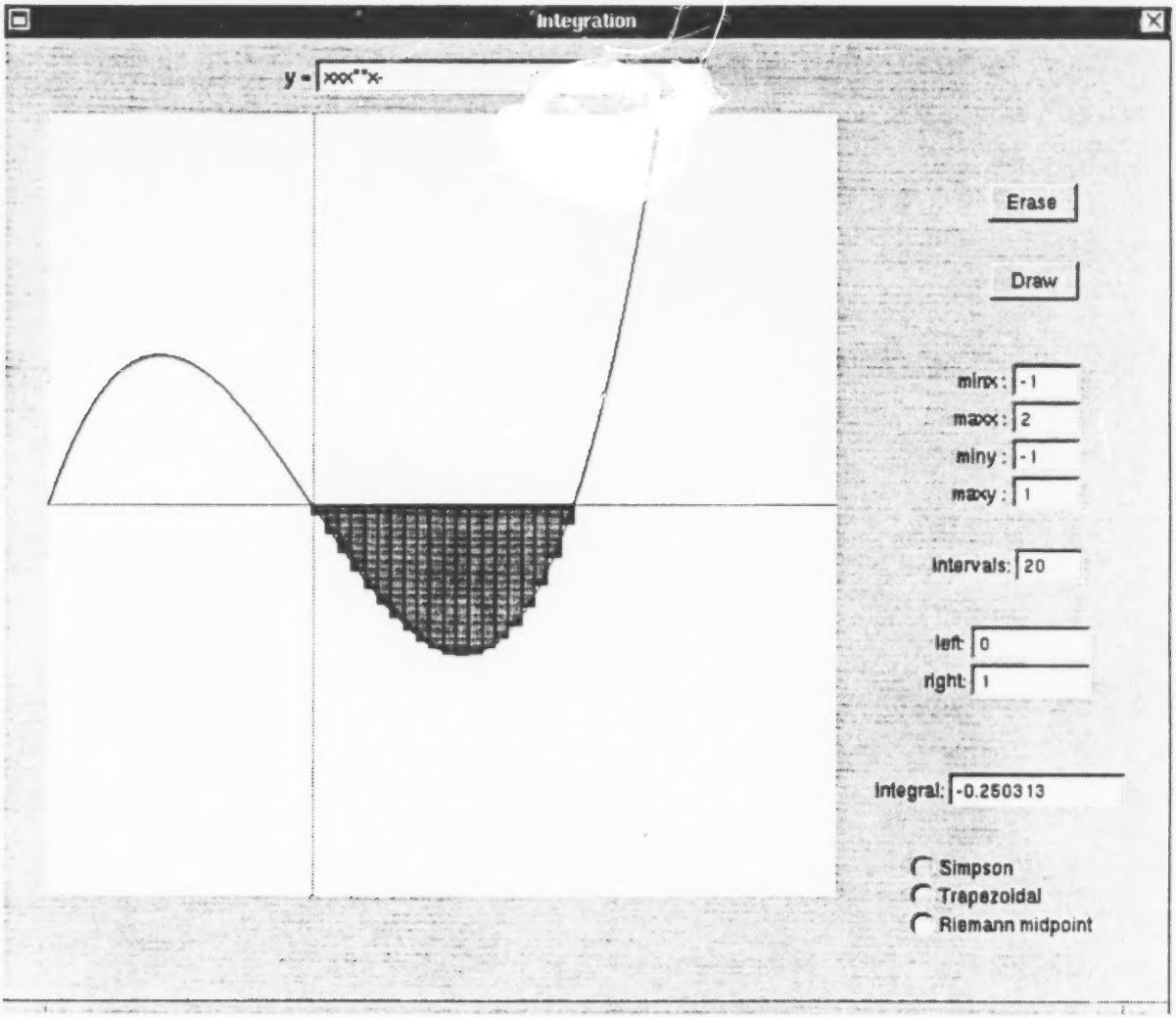
Don's new applications 3/14/90

The Polyhedron application rotates and shades the 5 regular (platonic) polyhedra, that is the tetrahedron, the hexahedron (cube), octahedron, the dodecahedron, and the icosahedron.



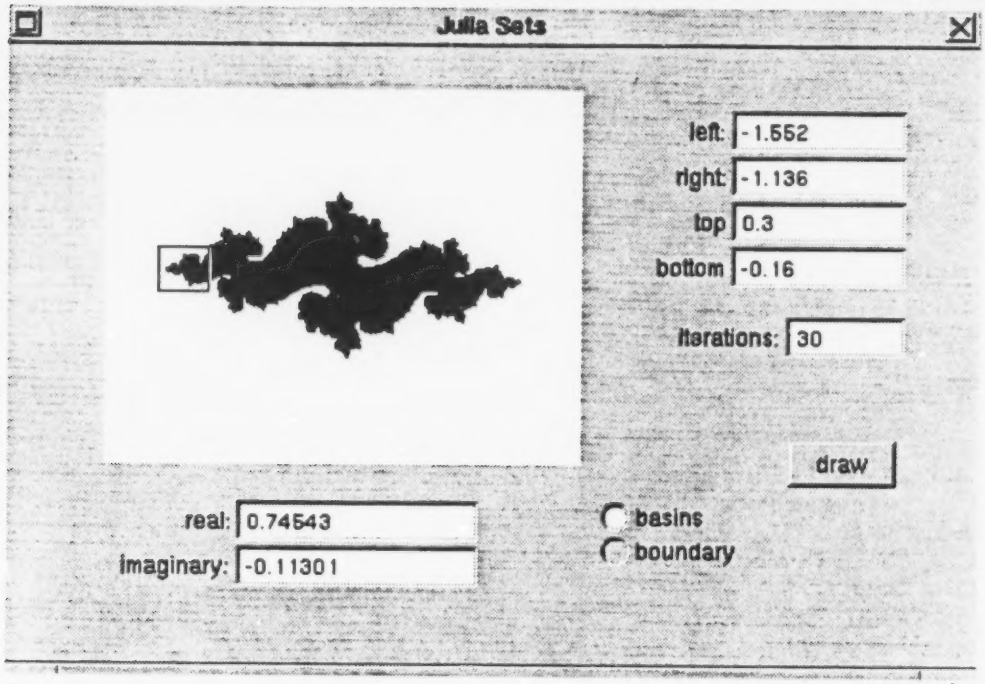
BEST COPY AVAILABLE

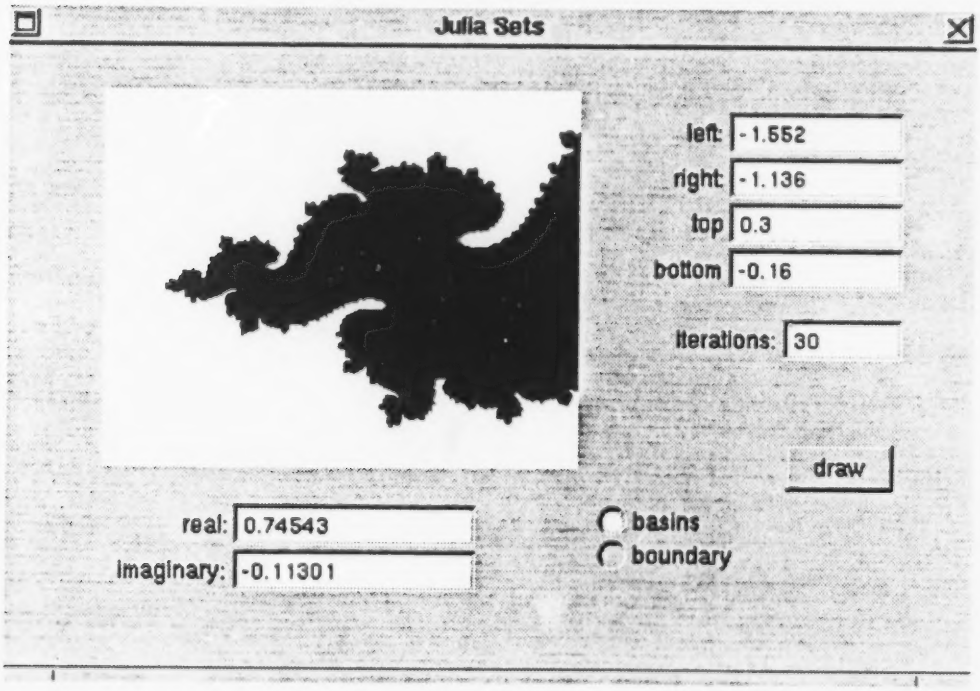
The Integration application allows the user to approximate the value of an integral of a function by Simpson's rule, the trapezoidal rule, or Riemann's midpoint sums.



BEST COPY AVAILABLE

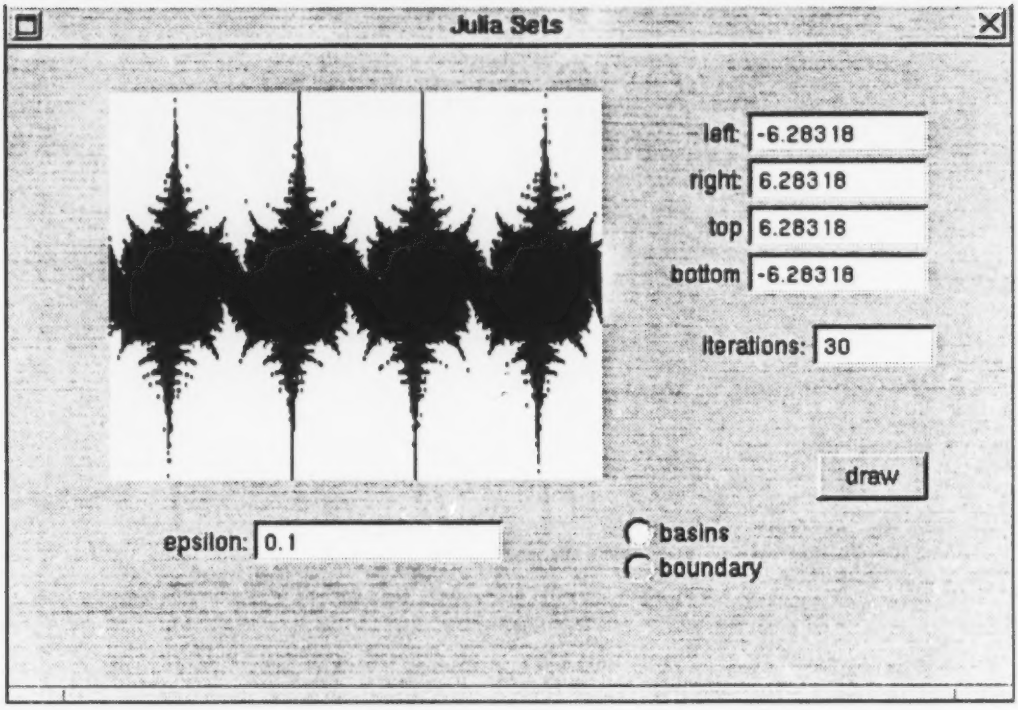
The Julia Set application can be used to draw different Julia Sets and zoom in on parts of the graph to display self-similarity. It can also display the boundary of the Julia Set.



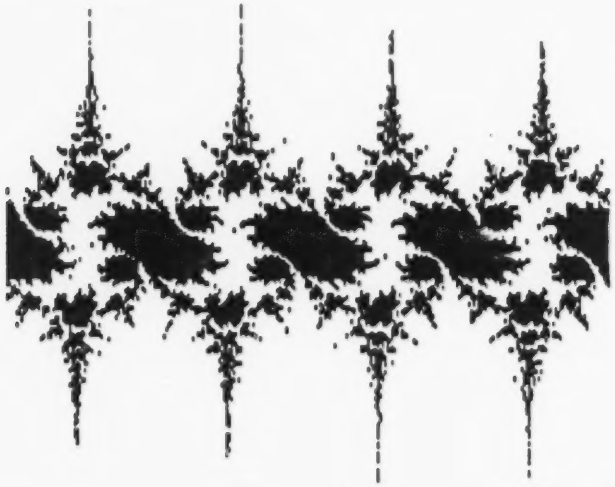


BEST COPY AVAILABLE

Another variation shows the Julia set of S(z) = (1+epsilon*i)*sin(z).

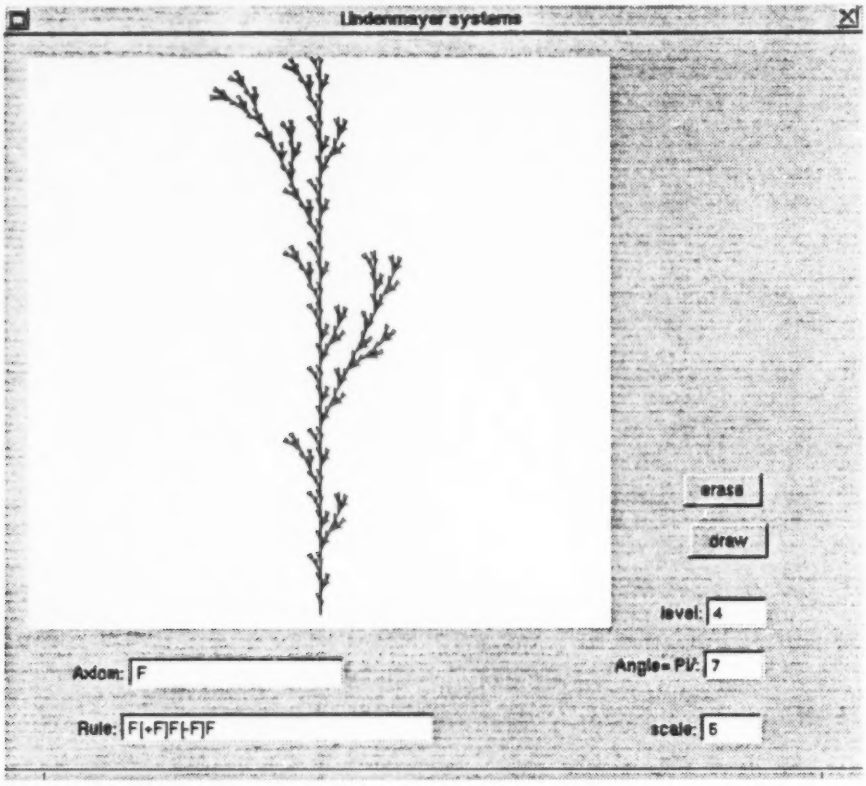


Changing the value of epsilon further yields such graphs as:

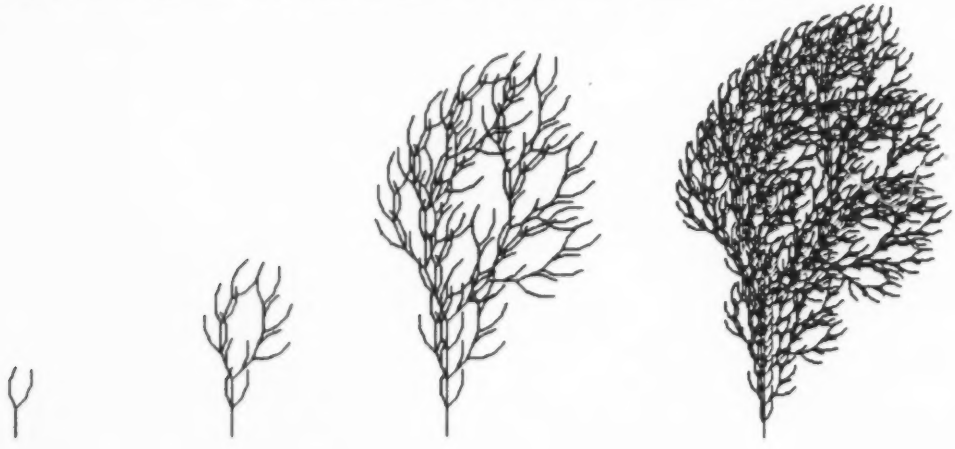


BEST COPY AVAILABLE

Lindenmayer systems allow the definition of natural looking objects to be described in formal language.



For example, one simple axiom and one relatively short production rule can be used to generate four generations of the following "bush". (These objects are sometimes called "graftals").



BEST COPY AVAILABLE